SANDIA REPORT

SAND2008-2639
Unlimited Release
Printed April 2008

The Portals 4.0 Message Passing Interface

Rolf Riesen, Ron Brightwell, Kevin Pedretti, and Brian Barrett, Sandia National Laboratories
Keith Underwood, Intel Corporation
Arthur B. Maccabe, University of New Mexico,
Trammell Hudson, Rotomotion

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2008-2639
Unlimited Release
Printed April 2008

The Portals 4.0 Message Passing Interface

Rolf Riesen
Ron Brightwell
Kevin Pedretti
Brian Barrett
Scalable Computing Systems Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
rolf@cs.sandia.gov
bright@cs.sandia.gov
ktpedre@sandia.gov
bwbarre@sandia.gov

Keith Underwood
DEG Architecture and Planning
Intel Corporation
P.O. Box 5800
Albuquerque, NM 87185-1319
Keith.D.Underwood@intel.com

Arthur B. Maccabe
Computer Science Department
University of New Mexico
Albuquerque, NM 87131-1386
maccabe@cs.unm.edu

Trammell Hudson
c/o OS Research
1527 16th NW #5
Washington, DC 20036
hudson@osresearch.net

Abstract

This report presents a specification for the Portals 4.0 message passing interface. Portals 4.0 are intended to allow scalable, high-performance network communication between nodes of a parallel computing system. Portals 4.0 are well suited to massively parallel processing and embedded systems. Portals 4.0 represent an adaption of the data movement layer developed for massively parallel processing platforms, such as the 4500-node Intel TeraFLOPS machine. Version 3.0 of Portals runs on the Cplant cluster at Sandia National Laboratories, and version 3.3 is running on Cray's Red Storm system. Version 4.0 is targeted to the next generation of machines employing advanced network interface architectures to support enhanced offload capabilities.

Acknowledgments

Over the years, many people have helped shape, design, and write portals code. We wish to thank: Eric Barton, Peter Braam, Lee Ann Fisk, David Greenberg, Eric Hoffman, Gabi Istrail, Jeanette Johnston, Chu Jong, Clint Kaul, Mike Levenhagen, Kevin McCurley, Jim Otto, David Robboy, Mark Sears, Lance Shuler, Jim Schutt, Mack Stallcup, Todd Underwood, David van Dresser, Dena Vigil, Lee Ward, and Stephen Wheat.

People who were influential in managing the project were: Bill Camp, Ed Barsis, Art Hale, and Neil Pundit

While we have tried to be comprehensive in our listing of the people involved, it is very likely that we have missed at least one important contributor. The omission is a reflection of our poor memories and not a reflection of the importance of their contributions. We apologize to the unnamed contributor(s).

Contents

Appendix

List of Figures

List of Tables

List of Implementation Notes

Preface

In the early 1990s, when memory-to-memory copying speeds were an order of magnitude faster than the maximum network bandwidth, it did not matter if data had to go through one or two intermediate buffers on its way from the network into user space. This began to change with early massively parallel processing (MPP) systems, such as the nCUBE-2 and the Intel Paragon, when network bandwidth became comparable to memory bandwidth. An intermediate memory-to-memory copy now meant that only half the available network bandwidth was used.

Early versions of Portals solved this problem in a novel way. Instead of waiting for data to arrive and then copy it into the final destination, Portals, in versions prior to 3.0, allowed a user to describe what should happen to incoming data by using data structures. A few basic data structures were used like Lego^{tm} blocks to create more complex structures. The operating system kernel handling the data transfer read these structures when data began to arrive and determined where to place the incoming data. Users were allowed to create matching criteria and to specify precisely where data would eventually end up. The kernel, in turn, had the ability to DMA data directly into user space, which eliminated buffer space in kernel owned memory and slow memory-to-memory copies. We named that approach Portals Version 2.0. It was used until 2006 on the ASCI Red supercomputer, the first general-purpose machine to break the one teraflops barrier.

Although very successful on architectures with lightweight kernels, such as ASCI Red, Portals proved difficult to port to Cplant [Brightwell et al. 2000] with its full-featured Linux kernel. Under Linux, memory was no longer physically contiguous in a one-to-one mapping with the kernel. This made it prohibitively expensive for the kernel to traverse data structures in user space. We wanted to keep the basic concept of using data structures to describe what should happen to incoming data. We put a thin application programming interface (API) over our data structures. We got rid of some never-used building blocks, improved some of the others, and Portals 3.0 were born.

We defined the Version 3.0 API in Brightwell, Hudson, Riesen, and Maccabe (1999). Since then, Portals have gone through three revisions. The latest was Version 3.3 Riesen, Brightwell, Maccabe, Hudson, and Pedretti (2006). In the interim, the system context has changed significantly. Many newer systems are capable of offloading the vast majority of the Portals implementation to the newtork interface. Indeed, the rapid growth of bandwidth and available silicon area relative to the small decrease in memory latency has made it *desirable* to move latency sensitive tasks like Portals matching to dedicated hardware better suited to it. The implementation of Version 3.3 on ASC Red Storm (Cray XT3/XT4/XT5) illuminated many challenges that have arisen with these advances in technology. In this report, we document Version 4.0 as a response to two specific challenges discovered on Red Storm. Foremost, while the performance of I/O buses has improved dramatically, the latency to cross an I/O bus relative to the target message rates has risen dramatically. In addition, partitioned global address space (PGAS) models have risen in prominence and require lighter weight semantics to support them.

Nomenclature

ACK	Acknowledgement.
FM	Illinois Fast Messages.
AM	Active Messages.
API	Application Programming Interface. A definition of the functions and semantics provided by library of functions.
ASCI	Advanced Simulation and Computing Initiative.
ASC	Advanced Simulation and Computing.
ASCI Red	Intel Tflops system installed at Sandia National Laboratories. First general-purpose system to break one teraflop barrier.
CPU	Central Processing Unit.
DMA	Direct Memory Access.
EQ	Event Queue.
FIFO	First In, First Out.
FLOP	Floating Point OPeration. (Also FLOPS or flops: Floating Point OPerations per Second.)
GM	Glenn's Messages; Myricom's Myrinet API.
ID	Identifier
Initiator	A *process* that initiates a message operation.
IOVEC	Input/Output Vector.
LE	List Entry.
MD	Memory Descriptor.
ME	Matching list Entry.
Message	An application-defined unit of data that is exchanged between *processes*.
Message Operation	Either a *put* operation, which writes data to a *target*, or a *get* operation, which reads data from a *target*, or a *atomic* which updates data atomically.
MPI	Message Passing Interface.
MPP	Massively Parallel Processor.
NAL	Network Abstraction Layer.
NAND	Bitwise Not AND operation.
Network	A network provides point-to-point communication between *nodes*. Internally, a network may provide multiple routes between endpoints (to improve fault tolerance or to improve performance characteristics); however, multiple paths will not be exposed outside of the network.
NI	Abstract portals Network Interface.
NIC	Network Interface Card.
Node	A node is an endpoint in a *network*. Nodes provide processing capabilities and memory. A node may provide multiple processors (an SMP node) or it may act as a *gateway* between networks.
OS	Operating System.
PM	Message passing layer for SCoreD [Ishikawa et al. 1996].
POSIX	Portable Operating System Interface.
Process	A context of execution. A process defines a virtual memory context. This context is not shared with other processes. Several threads may share the virtual memory context defined by a process.
RDMA	Remote Direct Memory Access.
RMPP	Reliable Message Passing Protocol.

SMP	Shared Memory Processor.
SUNMOS	Sandia national laboratories/University of New Mexico Operating System.
Target	A *process* that is acted upon by a message operation.
TCP/IP	Transmission Control Protocol/Internet Protocol.
Teraflop	10^{12} flops.
Thread	A context of execution that shares a virtual memory context with other threads.
UDP	User Datagram Protocol.
UNIX	A multiuser, multitasking, portable OS.
VIA	Virtual Interface Architecture.

Chapter 1

Introduction

1.1 Overview

This document describes an application programming interface for message passing between nodes in a system area network. The goal of this interface is to improve the scalability and performance of network communication by defining the functions and semantics of message passing required for scaling a parallel computing system to two million cores or more. This goal is achieved by providing an interface that will allow a quality implementation to take advantage of the inherently scalable design of Portals[1].

This document is divided into several sections:

Section 1 – Introduction.
This section describes the purpose and scope of the portals API[2].

Section 2 – An Overview of the Portals 4.0 API.
This section gives a brief overview of the portals API. The goal is to introduce the key concepts and terminology used in the description of the API.

Section 3 – The Portals 4.0 API.
This section describes the functions and semantics of the portals API in detail.

Section 4 – The Semantics of Message Transmission.
This section describes the semantics of message transmission. In particular, the information transmitted in each type of message and the processing of incoming messages.

Appendix A – FAQ.
Frequently Asked Questions about Portals.

Appendix B – Portals Design Guidelines.
The guiding principles behind the portals design.

Appendix C – README-template.
A template for a README file to be provided by each implementation. The README describes implementation specific parameters.

Appendix D – Implementations.
A brief description of the portals 4.0 reference implementation and the implementations that run on Cray's XT3/XT4/XT5 Red Storm machine.

Appendix E – Summary of Changes.
A list of changes between versions since Version 3.3.

[1] The word Portals is a plural proper noun. We use it when we refer to the definition, design, version, or similar aspects of Portals.

[2] We use the lower case portals when it is used as an adjective; e.g., portals document, a (generic) portals address, or portals operations. We use the singular when we refer to a specific portal or its attributes; e.g., portal index, portal table, or a (specific) portal address.

1.2 Purpose

Existing message passing technologies available for supercomputer network hardware do not meet the scalability goals required by emerging massively parallel processing platforms that will have as many as two million processor cores. This greatly exceeds the capacity for which existing message passing technologies have been designed and implemented.

In addition to the scalability requirements of the network, these technologies must also be able to support a scalable, high performance implementation of the Message Passing Interface (MPI) [Message Passing Interface Forum 1994] standard as well as the various partitioned global address space (PGAS) models, such as unified parallel C (UPC), Co-Array Fortran (CAF), and SHMEM [Cray Research, Inc. 1994]. While neither MPI nor PGAS models impose specific scalability limitations, many message passing technologies do not provide the functionality needed to allow implementations of MPI to meet our scalability or performance goals.

The following are required properties of a network architecture to avoid scalability limitations:

- Connectionless – Many connection-oriented architectures, such as InfiniBand [Infiniband Trade Association 1999], VIA [Compaq, Microsoft, and Intel 1997] and TCP/IP sockets, have practical limitations on the number of peer connections that can be established. In large-scale parallel systems, any node must be able to communicate with any other node without costly connection establishment and tear down.
- Network independence – Many communication systems depend on the host processor to perform operations in order for messages in the network to be consumed. Message consumption from the network should not be dependent on host processor activity, such as the operating system scheduler or user-level thread scheduler. Applications must be able to continue computing while data is moved in and out of the application's memory.
- User-level flow control – Many communication systems manage flow control internally to avoid depleting resources, which can significantly impact performance as the number of communicating processes increases. While Portals provides building blocks to enable flow control (See Section 2.3), it is the responsibility of the application to manage flow control. An application should be able to provide final destination buffers into which the network can deposit data directly.
- OS bypass – High performance network communication should not involve memory copies into or out of a kernel-managed protocol stack. Because networks are now as fast as memory buses, data has to flow directly into user space.

The following are properties of a network architecture that avoids scalability limitations for an implementation of MPI:

- Receiver-managed – Sender-managed message passing implementations require a persistent block of memory to be available for every process, requiring memory resources to increase with job size.
- User-level bypass (application bypass) – While OS bypass is necessary for high performance, it alone is not sufficient to support the *progress rule* of MPI asynchronous operations. After an application has posted a receive, data must be delivered and acknowledged without further intervention from the application.
- Unexpected messages – Few communication systems have support for receiving messages for which there is no prior notification. Support for these types of messages is necessary to avoid flow control and protocol overhead.

1.3 Background

Portals were originally designed for and implemented on the nCUBE-2 machine as part of the SUNMOS (Sandia/UNM OS) [Maccabe et al. 1994] and Puma [Shuler et al. 1995] lightweight kernel development projects.

Portals went through three design phases [Riesen et al. 2005], with the most recent one being used on the 13000-node (38,400 cores) Cray Red Storm [Alverson 2003] that became the Cray XT3/XT4/XT5 product line. Portals have been very successful in meeting the needs of such large machines, not only as a layer for a high-performance MPI implementation [Brightwell and Shuler 1996], but also for implementing the scalable run-time environment and parallel I/O capabilities of the machine.

The third generation portals implementation was designed for a system where the work required to process a message was long relative to the round trip between the application and the Portals data structures; however, in modern systems where processing is offloaded onto the network interface, the time to post a receive is dominated by the round trip across the I/O bus. This latency has become large relative to message latency and per message overheads (gap). This limitation was exposed by implementations on the Cray Red Storm system. Version 4.0 of Portals addresses this problem by adding the concept of *unexpected messages* to Portals. The second limitation exposed on Red Storm was the relative weight of handling newer PGAS programming models. PGAS programming models do not need the extensive matching semantics required by MPI and I/O libraries and can achieve significantly lower latency and higher message throughput without matching. Version 4.0 of Portals adds a lightweight, non-matching interface to support these semantics as well as lightweight events and acknowledgments. Finally, version 4.0 of Portals reduces the overheads in numerous implementation paths by simplifying events, reducing the size of acknowledgments, and generally specializing interfaces to eliminate data that experience has shown to be unnecessary.

1.4 Scalability

The primary goal in the design of Portals is scalability. Portals are designed specifically for an implementation capable of supporting a parallel job running on a million processing cores or more. Performance is critical only in terms of scalability. That is, the level of message passing performance is characterized by how far it allows an application to scale and not by how it performs in micro-benchmarks (e.g., a two-node bandwidth or latency test).

The portals API is designed to allow for scalability, not to guarantee it. Portals cannot overcome the shortcomings of a poorly designed application program. Applications that have inherent scalability limitations, either through design or implementation, will not be transformed by Portals into scalable applications. Scalability must be addressed at all levels. Portals do not inhibit scalability and do not guarantee it either. No portals operation requires global communication or synchronization.

Similarly, a quality implementation is needed for Portals to be scalable. If the implementation or the network protocols and hardware underneath it cannot scale to one million nodes, then neither Portals nor the application can.

To support scalability, the portals interface maintains a minimal amount of state. By default, Portals provide reliable, ordered delivery of messages between pairs of processes. Portals are connectionless: a process is not required to explicitly establish a point-to-point connection with another process in order to communicate. Moreover, all buffers used in the transmission of messages are maintained in user space. The *target* process determines how to respond to incoming messages, and messages for which there are no buffers are discarded.

IMPLEMENTATION NOTE 1: No wire protocol

This document does not specify a wire protocol. Portals require a reliable communication layer. Whether that is achieved through software or hardware is up to the implementation. For example, for Red Storm two reliability protocols were implemented — one by Cray and one by Sandia [Brightwell et al. 2006].

IMPLEMENTATION NOTE 2:

Weak Ordering Semantics

The default ordering semantics for Portals messages only requires that messages are *started* in order at the target. The underlying implementation is free to deliver the *body* of two messages in whatever order is necessary. This provides additional flexibility to the underlying implementation. For example, the network can use a retransmission protocol on the wire that retransmits a portion of a lost message without violating ordering. Similarly, an implementation is free to use adaptive routing to deliver the body of the message. An implementation may, however, choose to provide stronger ordering than is required. For example, to simplify the implementation of a `shmem_fence()`, an implementation may choose to provide strict ordering of data at the target. In addition, an initiator may explicitly indicate that a message does not have to be ordered at the target using an option on the MD (see Section 3.10). There is also an issue with the ordering of data. When data arrives in a region described by a list entry that happens to overlap with a region described by a memory descriptor with an active operation, the ordering of data operations is undefined. Data is only available for transmit after the event corresponding to the arriving message has been posted. Thus, triggered operations are safe, since they do not trigger until the counting event is posted.

Discussion: *The specified ordering semantics of Portals is not sufficient to allow a* `shmem_fence()` *operation to be treated as a no-op. Specific implementations of Portals may* choose *to provide more strict ordering requirements, or a SHMEM implementation may promote* `shmem_fence()` *to* `shmem_quiet()`.

1.5 Communication Model

Portals combine the characteristics of both one-sided and two-sided communication. In addition to more traditional "put" and "get" operations, they define "matching put" and "matching get" operations. The destination of a *put* (or send) is not an explicit address; instead, messages target match list entries (potentially with an offset) using the Portals addressing semantics that allow the receiver to determine where incoming messages should be placed. This flexibility allows Portals to support both traditional one-sided operations and two-sided send/receive operations.

Portals allow the *target* to determine whether incoming messages are acceptable. A *target* process can choose to accept message operations from any specific process or can choose to ignore message operations from any specific process.

1.6 Zero Copy, OS Bypass, and Application Bypass

In traditional system architectures, network packets arrive at the network interface card (NIC), are passed through one or more protocol layers in the operating system, and are eventually copied into the address space of the application. As network bandwidth began to approach memory copy rates, reduction of memory copies became a critical concern. This concern led to the development of zero-copy message passing protocols in which message copies are eliminated or pipelined to avoid the loss of bandwidth.

A typical zero-copy protocol has the NIC generate an interrupt for the CPU when a message arrives from the network. The interrupt handler then controls the transfer of the incoming message into the address space of the

appropriate application. The interrupt latency, the time from the initiation of an interrupt until the interrupt handler is running, is fairly significant. To avoid this cost, some modern NICs have processors that can be programmed to implement part of a message passing protocol. Given a properly designed protocol, it is possible to program the NIC to control the transfer of incoming messages without needing to interrupt the CPU. Because this strategy does not need to involve the OS on every message transfer, it is frequently called "OS bypass." ST [Task Group of Technical Committee T11 1998], VIA [Compaq, Microsoft, and Intel 1997], FM [Lauria et al. 1998], GM [Myricom, Inc. 1997], PM [Ishikawa et al. 1996], and Portals are examples of OS bypass mechanisms.

Many protocols that support OS bypass still require that the application actively participates in the protocol to ensure progress. As an example, the long message protocol of PM requires that the application receive and reply to a request to put or get a long message. This complicates the runtime environment, requiring a thread to process incoming requests, and significantly increases the latency required to initiate a long message protocol. The portals message passing protocol does not require activity on the part of the application to ensure progress. We use the term "application bypass" to refer to this aspect of the portals protocol.

IMPLEMENTATION NOTE 3: User memory as scratch space

The portals API allows for user memory where data is being received to be altered (e.g. at the *target*, or in a reply buffer at the *initiator*. That means an implementation can utilize user memory as scratch space and staging buffers. Only after an operation succeeds and the event has been posted must the user memory reflect exactly the data that has arrived. The portals API explicitly prohibits modifying the the buffer passed into a *put*.

1.7 Faults

Given the number of components that we are dealing with and the fact that we are interested in supporting applications that run for very long times, failures are inevitable. The portals API recognizes that the underlying transport may not be able to successfully complete an operation once it has been initiated. This is reflected in the fact that the portals API reports an event indicating the successful completion of every operation. Completion events carry a flag which indicates whether the operation completed successfully or not.

Between the time an operation is started and the time that the operation completes (successfully or unsuccessfully), any memory associated with "receiving data" should be considered volatile. That is, the memory may be changed in unpredictable ways while the operation is progressing. Once the operation completes, the memory associated with the operation will not be subject to further modification (from this operation). Notice that unsuccessful operations may alter memory used to receive data in an essentially unpredictable fashion. Memory associated with transmitting data must not be modified by the implementation.

IMPLEMENTATION NOTE 4: Don't alter put or reply buffers

An implementation must not alter data in a user buffer that is used in a *put* or *reply* operation. This is independent of whether the operation succeeds or fails.

Chapter 2

An Overview of the Portals API

In this chapter, we give a conceptual overview of the portals API. The goal is to provide a context for understanding the detailed description of the API presented in the next section.

2.1 Data Movement

A portal represents an opening in the address space of a process. Other processes can use a portal to read (*get*), write (*put*), or perform an atomic operation on the memory associated with the portal. Every data movement operation involves two processes, the *initiator* and the *target*. The *initiator* is the process that initiates the data movement operation. The *target* is the process that responds to the operation by accepting the data for a *put* operation, replying with the data for a *get* operation, or updating a memory location for, and potentially responding with the result from, an *atomic* operation.

In this discussion, activities attributed to a process may refer to activities that are actually performed by the process or *on behalf of the process.* The inclusiveness of our terminology is important in the context of *application bypass.* In particular, when we note that the *target* sends a reply in the case of a get operation, it is possible that a reply will be generated by another component in the system, bypassing the application.

Figure 2.1 shows the graphical conventions used throughout this document. Some of the data structures created through the portals API reside in user space to enhance scalability and peformance, while others are kept in protected space for protection and to allow an implementation to place these structures into host or NIC memory. We use colors to distinguish between these elements.

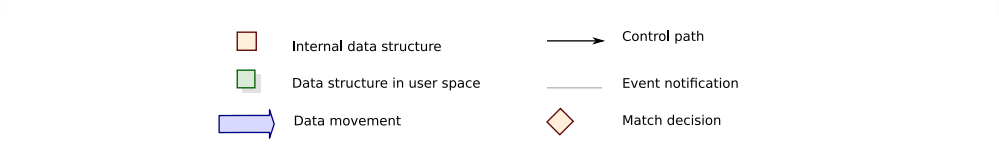


Figure 2.1. Graphical Conventions: Symbols, colors, and stylistic conventions used in the diagras of this document.

Figures 2.2, 2.3, 2.4, and 2.5 present graphical interpretations of the portals data movement operations: *put* (send), *get*, and *atomic* (atomic operation — swap is shown). In the case of a *put* operation, the *initiator* sends a put request ① message to the *target*. The *target* translates the portal addressing information in the request using its local portals structures. The data may be part of the same packet as the put request or it may be in separate packet(s) as shown in Figure 2.2. The portals API does not specify a wire protocol (Section 4). When the data ② has been put into the remote memory descriptor (or been discarded), the *target* optionally sends an acknowledgment ③ message.

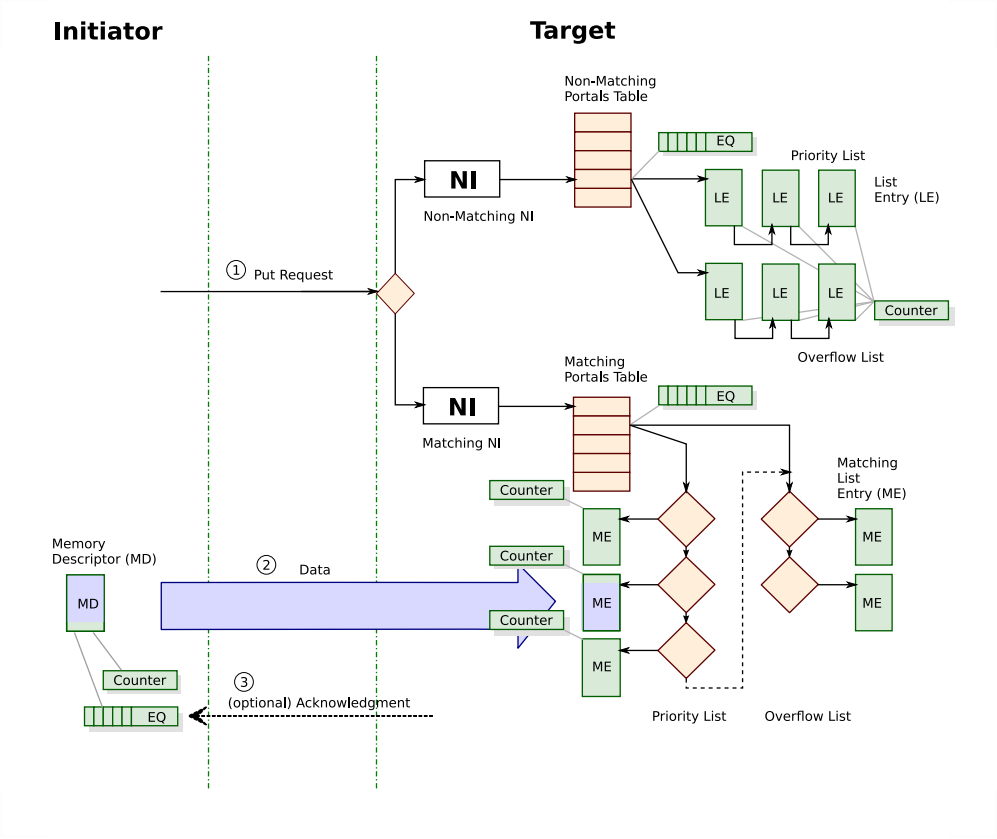


Figure 2.2. Portals Put (Send): Note that the put request ① is part of the header and the data ② is part of the body of a single message. Depending on the network hardware capabilities, the request and data may be sent in a single large packet or several smaller ones.

Figure 2.2 represents several important concepts in Portals 4.0. First, a message that arrives on one *physical* interface can nonetheless target multiple *logical* network interfaces. Figure 2.2 shows a *matching* and a *non-matching* network interface, but a given network interface can also use *logical* (rank) or *physical* (nid/pid) identifiers to refer to network endpoints (processes). As indicated in Figure 2.2, separate network interfaces have independent resources — even if they share a physical layer. The second important concept illustrated in Figure 2.2 is that each portal table entry has three data structures attached: an event queue, a priority list, and an overflow list. The final concept illustrated in Figure 2.2 is that the overflow list is traversed after the priority list. If a message does not match in the priority list (matching interface) or it is empty (either interface), the overflow list is traversed.

Figure 2.2 illustrates another important Portals concept. The space the Portals data structures occupy is divided into protected and application (user) space, while the large data buffers reside in user space. Most of the portals data structures reside in protected space. Often the portals control structures reside inside the operating system kernel or the network interface card. However, they can also reside in a library or another process. See implementation note 5 for possible locations of the event queues.

IMPLEMENTATION NOTE 5: Location of event queues and counters

Note that data structures that can only be accessed through the API, such as counters and event queues, are intended to reside in user space. However, an implementatin is free to place them anywhere it wants.

IMPLEMENTATION NOTE 6: Protected space

Protected space as shown for example in Figure 2.2 does not mean it has to reside inside the kernel or a different address space. The portals implementation must guarantee that no alterations of portals structures by the user can harm another process or the portals implementation.

Figure 2.3 is a representation of a *get* operation from a *target* that does matching. The corresponding *get* from a non-matching *target* is shown in Figure 2.4. First, the *initiator* sends a request ① to the *target*. As with the *put* operation, the *target* translates the portals addressing information in the request using its local portals structures. Once it has translated the portals addressing information, the *target* sends a *reply*② that includes the requested data.

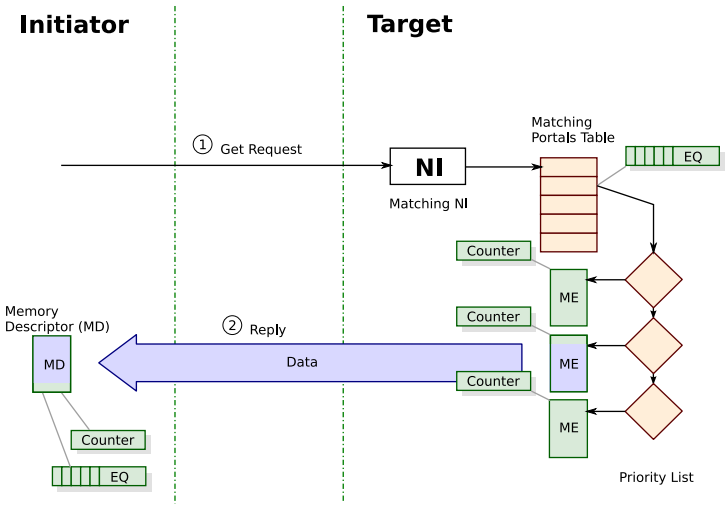


Figure 2.3. Portals Get from a match list entry.

We should note that portals address translations are only performed on nodes that respond to operations initiated by other nodes; i.e., a *target*. Acknowledgments for *put* operations and replies to *get* and *atomic* operations bypass the portals address translation structures at the *initiator*.

The third operation, *atomic* (atomic operation), is depicted in Figure 2.5 for the swap operation and Figure 2.6 for a summation.

For the swap operation shown in Figure 2.5, the *initiator* sends a request ①, containing the *put* data and the operand value ②, to the *target*. The *target* traverses the local portals structures based on the information in the request to find the appropriate user buffer. The *target* then sends the *get* data in a *reply* message ③ back to the *initiator* and deposits the *put* data in the user buffer.

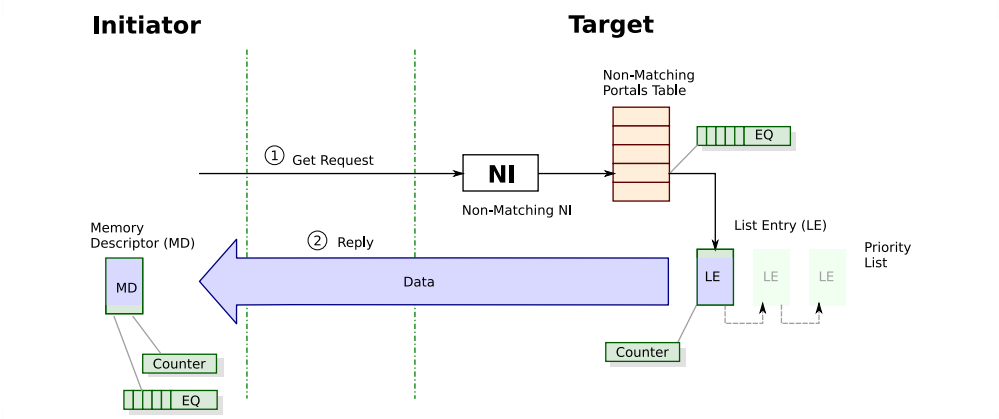


Figure 2.4. Portals Get from a list entry. Note that the first LE will be selected to reply to the *get* request.

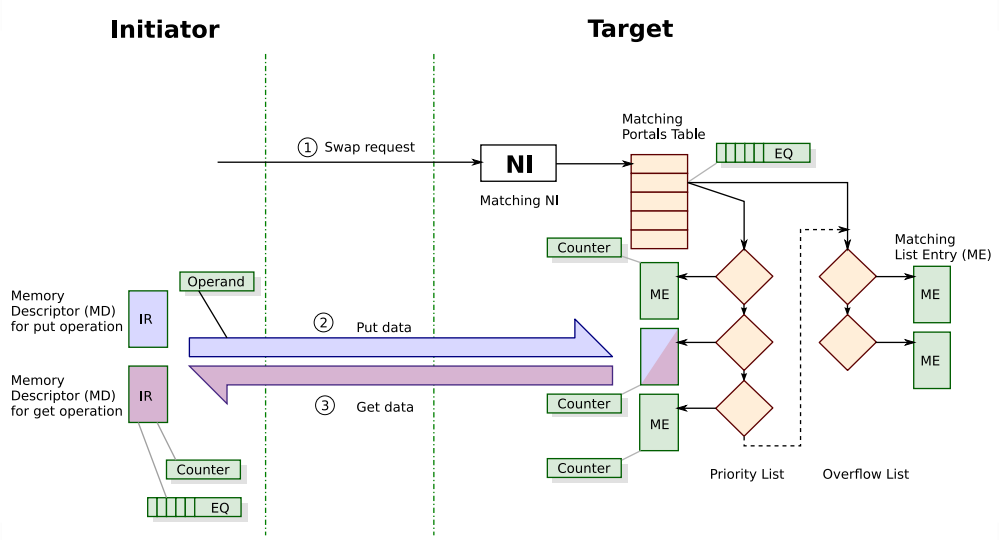


Figure 2.5. Portals Atomic (swap is shown). An atomic swap in memory described by a match list entry using an initiator-side operand.

The sum operation shown in Figure 2.6 adds the put data into the memory region described by the list entry. The figure shows an optional *acknowledgment* sent back. The result of the summation is not sent back, since the *initiator* used **PtlAtomic()** instead of **PtlFetchAtomic()**.

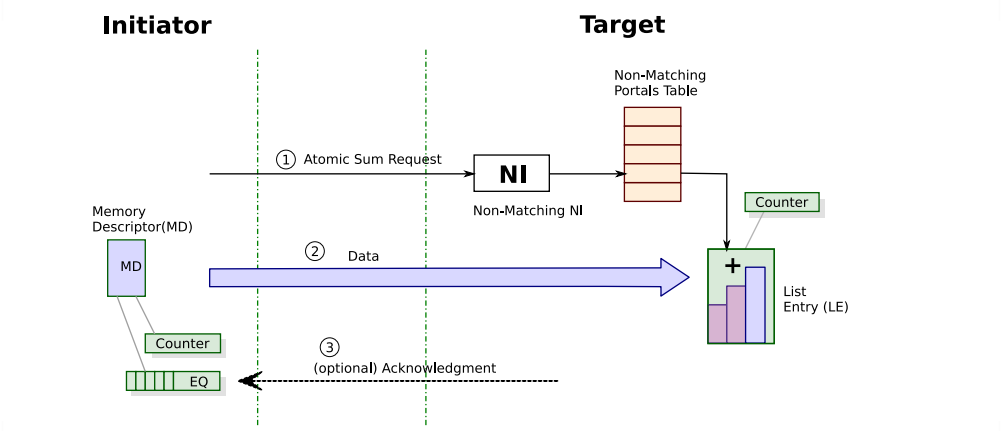


Figure 2.6. Portals Atomic (sum is shown). An atomic sum operation in memory described by a list entry.

2.2 Portals Addressing

One-sided data movement models (e.g., shmem [Cray Research, Inc. 1994], ST [Task Group of Technical Committee T11 1998], and MPI-2 [Message Passing Interface Forum 1997]) typically use a triple to address memory on a remote node. This triple consists of a process identifier, memory buffer identifier, and offset. The process identifier identifies the *target* process, the memory buffer identifier specifies the region of memory to be used for the operation, and the offset specifies an offset within the memory buffer.

In addition to the standard address components (process identifier, memory buffer identifier, and offset), a portals address can include information identifying the *initiator* (source) of the message and a set of match bits. This addressing model is appropriate for supporting one-sided operations, as well as traditional two-sided message passing operations. Specifically, the portals API provides the flexibility needed for an efficient implementation of MPI-1, which defines two-sided operations with one-sided completion semantics.

Once the target buffer has been selected, the incoming message must pass a permissions check. The permissions check is *not* a component of identifying the correct buffer. It is *only* applied once the correct buffer has been identified. The permissions check has two components: the sender of the message must be allowed to access this buffer, and the operation type selected must be allowed. Each list entry and match list entry has specifiers of which types of operations are allowed — put and/or get — as well as either a user ID or a job ID that can be used to identify which initiators are allowed to access the buffer. A failure in the permissions check does not modify the Portals state in any way, except to update the status registers (see Section 3.5.4).

Figures 2.7 and 2.8 are graphical representation of the structures used by a *target* in the interpretation of a portals address. The node identifier is used to route the message to the appropriate node and is not reflected in this diagram. The process ID[1] process identifier is used to select the correct *target* process and the network interfaces it has initialized. The network interface used by the initiator is used to select the correct portal table. There is one portal table for each process and each interface initialized by the process; i.e., if a process initializes an interface for a Myrinet and then initializes another interface for an Ethernet, two portal tables will be created within that process, one for each interface. Similarly, if one physical interface has been initialized as a matching interface and is later initialized as a non-matching interface, each logical interface has an independent portal table. Figure 2.7 shows the flow of addressing information in the case of an unmatched NI, while Figure 2.8 illustrates the case of a matched data

[1] A logical *rank* can be substituted for the combination of node ID and process ID when logical end-point addressing is used.

transfer.

The portal index is used to select an entry in the portal table. Each entry of the portal table identifies two lists and, optionally, an event queue. The first list is a priority list that is posted by the application to describe remotely accessible address regions. If matching is enabled for the selected network interface, each element of the priority list specifies two bit patterns: a set of “don’t care” bits and a set of “must match” bits. Along with source node ID (NID) and source process ID (PID), these bits are used in a matching function to select the correct match list entry. If matching is not enabled, the first entry in the list is used. The second list associated with each portal table entry is an overflow list. The overflow list maintains (loosely) the same semantics as the priority list. If the network interface provides matching on the priority list, then it provides it on the overflow list. If the network interface is configured to be non-matching, then the overflow list does not provide matching. The overflow list is always traversed *after* the priority list. It uses locally managed offsets to provide a space for the Portals implementation to store unexpected messages, and any associated state that the implementation deems necessary. The application populates the overflow list with either list entries (non-matching network interface) or match list entries (matching network interface) that are used and then unlinked by the implementation. An overflow list entry is not *required* to have a buffer associated with it, since the overflow list semantics allow the application to post a list entry that drops the body of messages; however, if the portal table entry has enabled flow control, then exhaustion of the overflow list will lead to a `PTL_EVENT_PT_DISABLED` being posted at the target when a message arrives.

List entries identify a memory region as well as an optional counting event. Matching list entries add a set of matching criteria to this identifier. For both the list entries and match list entries, the application can specify a set of protection criteria. The protection criteria includes the type of operations allowed (put and/or get) as well as who is allowed to access the buffer (either user ID, job ID, or a wildcard). The memory region specifies the memory to be used in the operation, and the counting event is used to record the occurrence of operations. Information about the operations is (optionally) recorded in the event queue attached to the portal table entry.

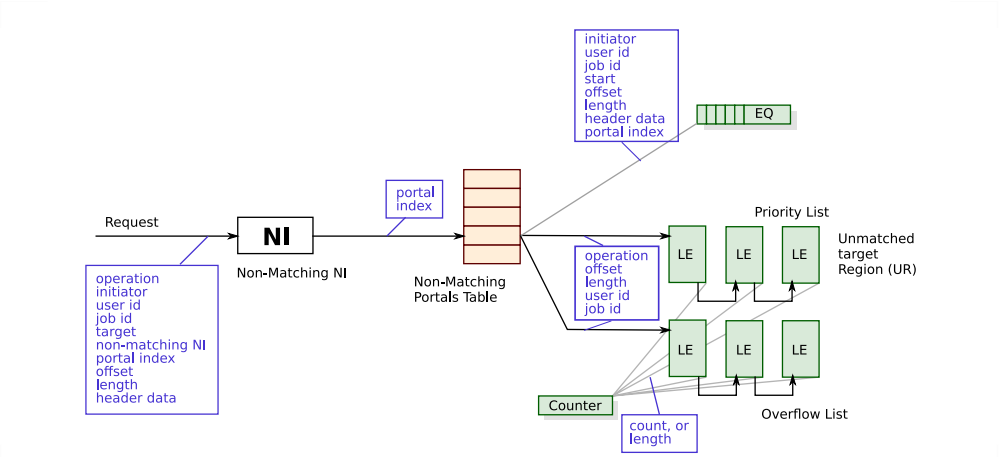


Figure 2.7. Portals Non-Matching Addressing Structures: The example shows the flow of information for an unmatched request at a target. Various pieces of information from the incoming header flow to the portals structures where they are needed to process the request.

Figure 2.9 illustrates the steps involved in translating a portals address when matching is enabled, starting from the first element in a priority list. If the match criteria specified in the match list entry are met, the permissions check passes, and the match list entry accepts the operation[2], the operation (*put*, *get*, or *atomic*) is performed using the

[2]Even if an incoming message matches the match criteria of a match list entry, the match list entry can reject operations because the memory region does not have sufficient space or the wrong operation is attempted. See Section 3.10.

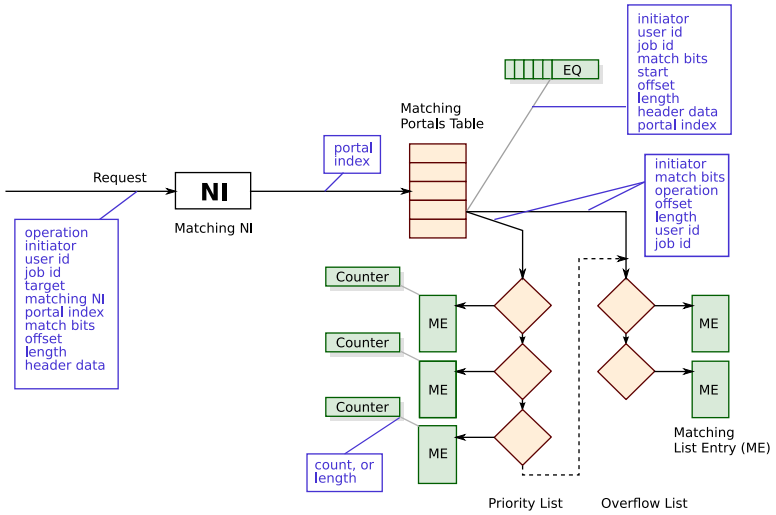


Figure 2.8. Portals Matching Addressing Structures: The example shows the flow of information for a matched request at a target. Various pieces of information from the incoming header flow to the portals structures where they are needed to process the request.

memory region specified in the match list entry. Note that matching is done using the match bits, ignore bits, node identifier, and process identifier.

If the match list entry specifies that it is to be unlinked based on the *min_free* semantic or if it is a use once match list entry, the match list entry is removed from the match list, and the resources associated with the match list entry are reclaimed. If there is an event queue specified in the portal table entry and the match list entry accepts the event, the operation is logged in the event queue. An event is written when no more actions, as part of the current operation, will be performed on this match list entry.

If the match criteria specified in the match list entry are not met, the address translation continues with the next match list entry. In contrast, if the permissions check fails or the match list entry rejects the operation, the matching ceases and the message is dropped without modifying the list state. If the end of the priority list has been reached, the address translation continues with the overflow list. The overflow list contains a series of buffers provided by the host for use by the implementation for messages that do not match in the priority list. The Portals implementation can capture the entire message, or any portion thereof allowed by the parameters of the match list entry. If a later match list entry is posted that matches an item in the overflow list, the implementation delivers an event (`PTL_EVENT_PUT_OVERFLOW`) to the application that includes a start address (which can be NULL) pointing to the location of the message. If the *rlength* and *mlength* in the event are equal, the start address must be a valid address indicating the location where the message arrived. If the *mlength* is less than the *rlength*, the message was truncated. This only occurs when the application has configured match list entries to discard message bodies; thus, the application is responsible for implementing the protocol necessary to retrieve the message body. If the overflow list does not have sufficient space for the message, the incoming request is discarded and a `PTL_EVENT_DROPPED` event is posted to the event queue associated with the portal table entry.

Discussion: *While overflow list semantics are convenient for managing unexpected messages, they do provide the potential for the implementation to push data movement onto the application when unexpected messages arrive. This makes it difficult, perhaps even impossible, for the implementation to*

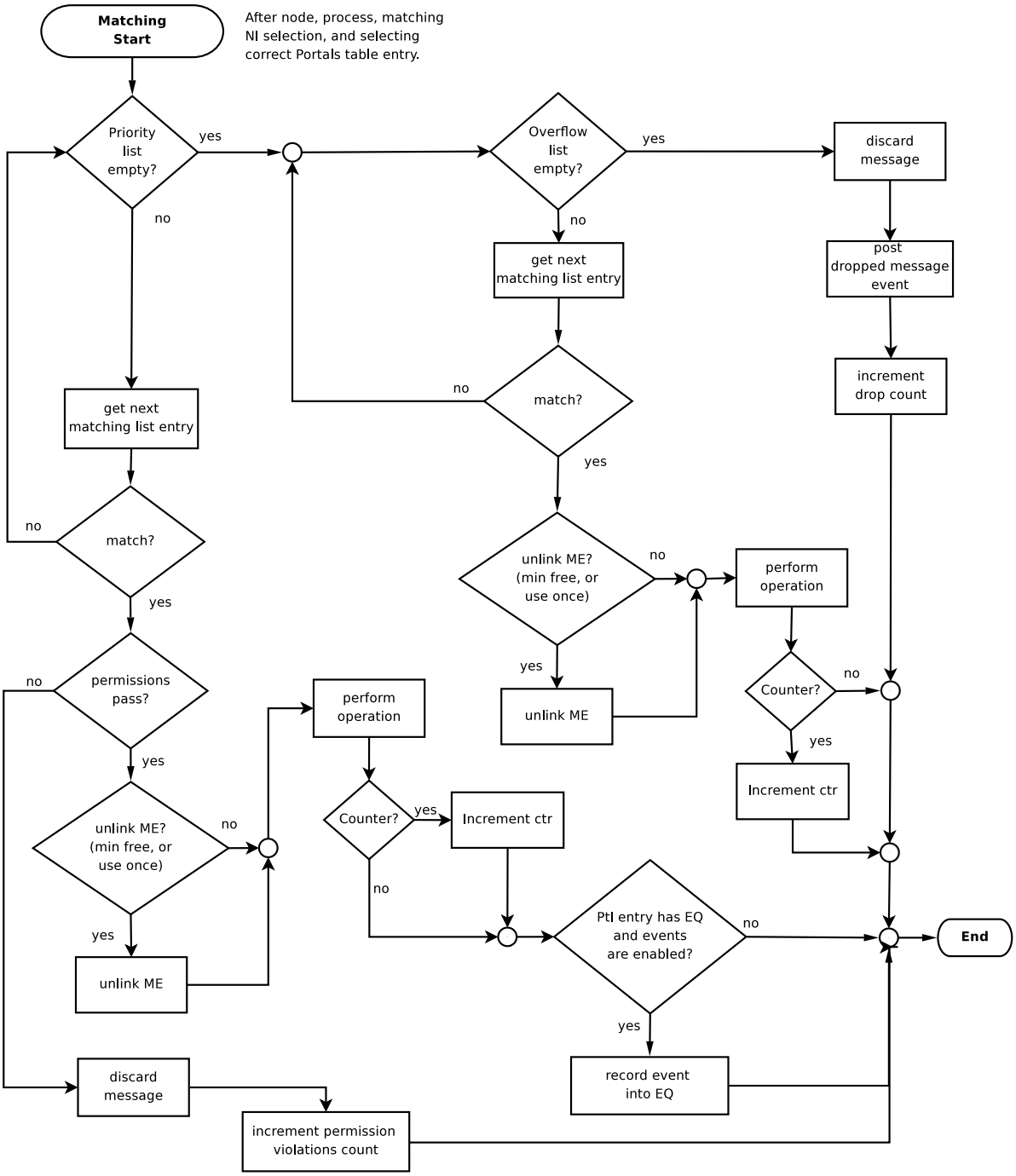


Figure 2.9. Matching Portals Address Translation.

know when the data movement associated with those messages is completed. While this does not change the ordering semantics of Portals, it highlights a subtlety that can be easily overlooked: Portals only guarantees that messages start *in order. Portals does not guarantee that messages complete in order; thus, a* ***PtlGet()*** *that follows a* ***PtlPut()*** *is not guaranteed to return the data delivered by the* ***PtlPut()*** *unless other, higher level ordering semantics are enforced. Similarly, when data arrives in a region described by*

a list entry that happens to overlap with a region described by a memory descriptor with an active operation, the ordering of data operations is undefined. Data is only available for transmit after the event corresponding to the arriving message has been posted. Thus, triggered operations are safe, since they do not trigger until the counting event is posted.

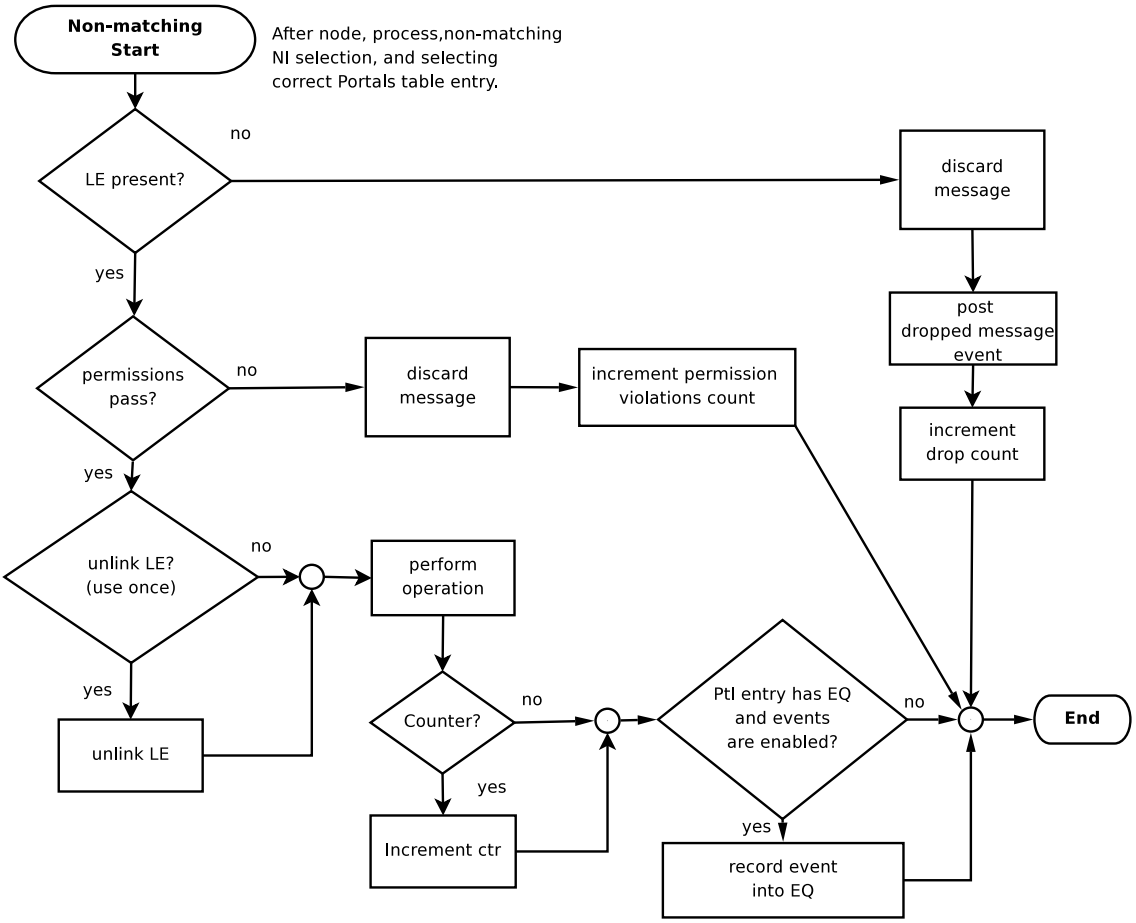


Figure 2.10. Non-Matching Portals Address Translation.

Figure 2.10 shows the comparable figure for address translation on a non-matching network interface. If matching is disabled, the portals address translation is dramatically simplified. The first list entry (LE) *always* matches. Authentication is provided through fields associated with the LE and act as *permission* fields, which can cause the operation to fail. An operation can fail to fit in the region provided and, if so, will be truncated; however, other semantics, such as locally managed offsets are not supported on the priority list when matching is not enabled. Locally managed offsets are always used in the overflow list. The overflow list is checked after the priority list, if necessary. If no list entry is present, the message is discarded and a `PTL_EVENT_DROPPED` event is posted. The non-matching translation path has the same event semantics as a matching interface. The important difference between the non-matching interface and the matching interface is that the address translation semantics for the non-matching interface (shown in Figure 2.10) have no loops. This allows fully pipelined operation for the non-matching address translation.

In typical scenarios, MPI uses the matching interface and requests full events in the event queue. SHMEM would use the non-matching interface and request only counting events be enabled at the initiator and no events be delivered at the target. In this mode, significantly lighter weight semantics can be delivered for PGAS style messaging, while full

offloading and independent progress can be guaranteed for MPI.

IMPLEMENTATION NOTE 7:

Overflow list

The overflow list can be managed in a number of ways; however, the most obvious implementation would use a locally managed offset and retain entire short messages or headers only for long messages (by posting a match list entry without a buffer and setting it to truncate). The implementation is neither *required to* or *prohibited from* using any space provided by match list entries in the overflow list to store message headers; however, the application is not required to provide such space with a match list entry. Thus, the implementation must have (or be able to acquire) state of its own. It may choose to augment that state with the space provided with the match list entries to store message headers. An implementation should *never* place information relating to one message into two different list entries as this will bind both entries until a matching match list entry is attached.

IMPLEMENTATION NOTE 8:

Non-matching address translation

A quality implementation would optimize for the common case of always using the head of the list for non-matching address translation. This could allow extremely high message rates for non-matching operations.

2.3 Flow Control

Historically, on some large machines, MPI over Portals has run into problems where the number of unexpected messages has caused the exhaustion of event queue space and buffer buffer set aside for unexpected messages. While this level of unexpected messages is an example of truly terrible programming, nonetheless it is a behavior that commercial MPI implementations encounter. In the past, this has caused the loss of an event or a message and the MPI application is lost. Users then complain. As an example of how other networks solve this issue, InfiniBand uses "receiver not ready" NACKs and retransmits at the hardware level. Unfortunately, this is known to prohibit parallelism in the NIC and is detrimental to InfiniBand performance in some areas.

In attempting to address this challenge, Portals adopts the philosophy that such behavior will lead to extremely slow application performance anyway. Thus, if the application causes exhaustion of resources, recovery from this condition can be very slow. It must, however, be possible to recover.

When resources are exhausted, whether they are user allocated resources like EQ entries or implementation level resources, the implementation may choose to block new message processing for a constrained amount of time. If the resources remain exhausted, the implementation must disable the portal table entry and deliver an event to the application. At this point, all messages targeting that portal table entry for that process must be dropped until **PtlPTEnable()** is called. Note that a *reply* does not target a portal table entry and is not dropped. In addition, the `PTL_EVENT_SEND` event associated with that message (and subsequent in flight messages) fails with an appropriate indication in the *ni_fail_type* variable. The application (e.g. MPI library) must then use a second portal table entry to recover from the overflow. Recovery is painful — the user must quiesce the library (e.g. MPI), ensure that resources are available, re-enable the portal table entry, and restart communications. Quiescing the library requires the MPI libraray to insure that no more messages are in flight targeting the node that has experienced resource exhaustion. Making resources available involves draining all events from the event queue associated with the portal table entry,

replenishing the user allocated buffers on the overflow list, and draining unexpected messages from the Portals implementation.

2.4 Multi-Threaded Applications

The portals API supports a generic view of multi-threaded applications. From the perspective of the portals API, an application program is defined by a set of processes. Each process defines a unique address space. The portals API defines access to this address space from other processes (using portals addressing and the data movement operations). A process may have one or more *threads* executing in its address space.

With the exception of **PtlEQWait()** and possibly **PtlEQPoll()**, every function in the portals API is non-blocking and atomic with respect to both other threads and external operations that result from data movement operations. While individual operations are atomic, sequences of these operations may be interleaved between different threads and with external operations. The portals API does not provide any mechanisms to control this interleaving. It is expected that these mechanisms will be provided by the API used to create threads.

2.5 Usage

Some of the diagrams presented in this chapter may seem daunting at first sight. However, many of the diagrams show all possible options and features of the Portals building blocks. In actual use, only some of them are needed to accomplish a given function. Rarely will they all be active and used at the same time.

Figure 2.2 shows the complete set of options available for a *put* operation. In practice, a diagram like Figure 2.11 is much more realistic. It shows the Portals structures used to setup a one-sided *put* operation. A user of Portals needs to specify an initiator region where the data is to be taken from, and an unmatched target region to put the data. Offsets can be used to address portions of each region; e.g., a word at a time, and an event queue or an event counter inform the user when an individual transfer has completed.

Another example is Figure 2.6 which is simpler than Figure 2.5 and probably more likely to be used. Atomic operations, such as the one in Figure 2.6 are much more likely to use a single unmatched target region. Such simple constructs can be used to implement global reference counters, or access locks.

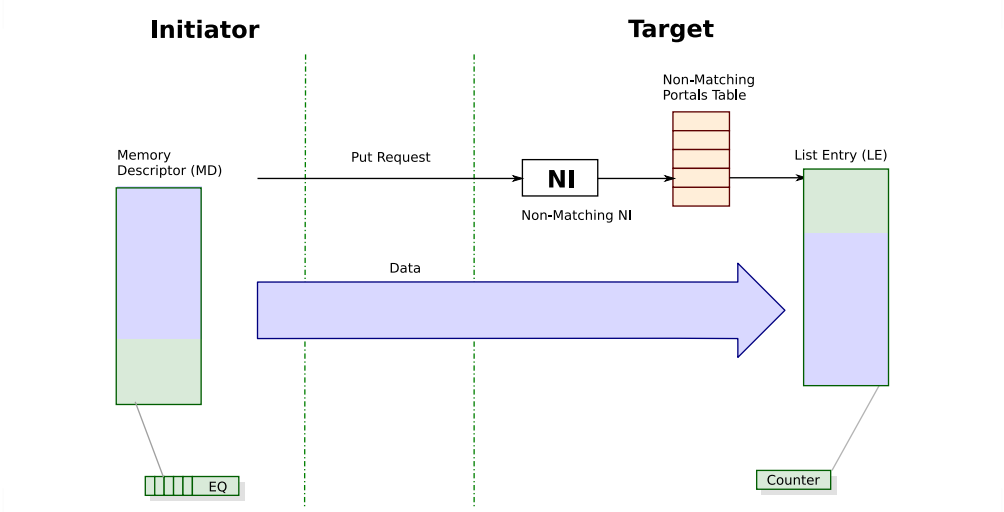


Figure 2.11. Simple Put Example: Not every option or Portals features is needed to accomplish simple tasks such as the transfer of data from an initiator region to a target region.

Chapter 3

The Portals API

3.1 Naming Conventions and Typeface Usage

The portals API defines four types of entities: functions, types, return codes, and constants. Functions always start with **Ptl** and use mixed upper and lower case. When used in the body of this report, function names appear in sans serif bold face, e.g., **PtlInit()**. The functions associated with an object type will have names that start with **Ptl**, followed by the two letter object type code shown in column *yy* in Table 3.1. As an example, the function **PtlEQAlloc()** allocates resources for an event queue.

Table 3.1. Object Type Codes.

yy	*xx*	Name	Section
NI	ni	Network Interface	3.5
PT	pt	Portal Table Entry	3.6
LE	le	List Entry	3.11
ME	me	Matching list Entry	3.12
MD	md	Memory Descriptor	3.10
EQ	eq	Event Queue	3.13
CT	ct	Count	3.14

Type names use lower case with underscores to separate words. Each type name starts with **ptl_** and ends with **_t**. When used in the body of this report, type names appear like this: **ptl_match_bits_t**.

Return codes start with the characters **PTL_** and appear like this: **PTL_OK**.

Names for constants use upper case with underscores to separate words. Each constant name starts with `PTL_`. When used in the body of this report, constant names appear like this: `PTL_ACK_REQ`.

The definition of named constants, function prototypes, and type definitions must be supplied in a file named `portals4.h` that can be included by programs using portals.

IMPLEMENTATION NOTE 9:

README and portals4.h

Each implementation must supply an include file named `portals4.h` with the definitions specified in this document. There should also be a README file that explains implementation specific details. For example, it should list the limits (Section 3.5.1) for this implementation and provide a list of status registers that are provided (Section 3.2.7). See Appendix C for a template.

3.2 Base Types

The portals API defines a variety of base types. These types represent a simple renaming of the base types provided by the C programming language. In most cases these new type names have been introduced to improve type safety and to avoid issues arising from differences in representation sizes (e.g., 16-bit or 32-bit integers). Table 3.3 lists all the types defined by Portals.

3.2.1 Sizes

The type **ptl_size_t** is an unsigned 64-bit integral type used for representing sizes.

3.2.2 Handles

Objects maintained by the API are accessed through handles. Handle types have names of the form `ptl_handle_`*xx*`_t`, where *xx* is one of the two letter object type codes shown in Table 3.1, column *xx*. For example, the type **ptl_handle_ni_t** is used for network interface handles. Like all portals types, their names use lower case letters and underscores are used to separate words.

Each type of object is given a unique handle type to enhance type checking. The type **ptl_handle_any_t** can be used when a generic handle is needed. Every handle value can be converted into a value of type **ptl_handle_any_t** without loss of information.

Handles are not simple values. Every portals object is associated with a specific network interface and an identifier for this interface (along with an object identifier) is part of the object handle.

IMPLEMENTATION NOTE 10: Network interface encoded in handle

Each handle must encode the network interface it is associated with.

IMPLEMENTATION NOTE 11: Size of handle types

It is highly recommended that a handle type should be no larger than the native machine word size.

The constant `PTL_EQ_NONE`, of type **ptl_handle_eq_t**, is used to indicate the absence of an event queue. Similarly, the constant `PTL_CT_NONE`, of type **ptl_handle_ct_t**, indicates the absence of a counting type event. See Section 3.10.1 for uses of these values. The special constant `PTL_INVALID_HANDLE` is used to represent an invalid handle.

3.2.3 Indexes

The type **ptl_pt_index_t** is an integral type used for representing portal table indexes. See Section 3.5.1 and 3.5.2 for limits on values of this type.

3.2.4 Match Bits

The type **ptl_match_bits_t** is capable of holding unsigned 64-bit integer values.

3.2.5 Network Interfaces

The type **ptl_interface_t** is an integral type used for identifying different network interfaces. Users will need to consult the implementation documentation to determine appropriate values for the interfaces available. The special constant PTL_IFACE_DEFAULT identifies the default interface.

3.2.6 Identifiers

The type **ptl_nid_t** is an integral type used for representing node identifiers and **ptl_pid_t** is an integral type for representing process identifiers when physical addressing is used in the network interface (PTL_NI_PHYSICAL is set for the network interface). If PTL_NI_LOGICAL is set, a *rank* (**ptl_rank_t**) is used instead. **ptl_uid_t** is an integral type for representing user identifiers, and **ptl_jid_t** is an integral type for representing job identifiers.

The special values PTL_PID_ANY matches any process identifier, PTL_NID_ANY matches any node identifier, PTL_RANK_ANY matches any rank, PTL_UID_ANY matches any user identifier, and PTL_JID_ANY matches any job identifier. See Section 3.11 and 3.12 for uses of these values.

3.2.7 Status Registers

Each network interface maintains an array of status registers that can be accessed using the **PtlNIStatus()** function (Section 3.5.4). The type **ptl_sr_index_t** defines the types of indexes that can be used to access the status registers. Only two indexes are defined for all implementations: PTL_SR_DROP_COUNT, which identifies the status register that counts the dropped requests for the interface, and PTL_SR_PERMISSIONS_VIOLATIONS, which counts the number of attempted permission violations. Other indexes (and registers) may be defined by the implementation.

The type **ptl_sr_value_t** defines the types of values held in status registers. This is a signed integer type. The size is implementation dependent but must be at least 32 bits.

3.3 Return Codes

The API specifies return codes that indicate success or failure of a function call. In the case where the failure is due to invalid arguments being passed into the function, the exact behavior of an implementation is undefined. The API suggests error codes that provide more detail about specific invalid parameters, but an implementation is not required to return these specific error codes. For example, an implementation is free to allow the caller to fault when given an invalid address, rather than return **PTL_SEGV**. In addition, an implementation is free to map these return codes to standard return codes where appropriate. For example, a Linux kernel-space implementation could map portals return codes to POSIX-compliant return codes. Table 3.5 lists all return codes used by Portals.

3.4 Initialization and Cleanup

The portals API includes a function, **PtlInit()**, to initialize the library and a function, **PtlFini()**, to clean up after the process is done using the library. The initialization state of Portals is reference counted so that repeated calls to **PtlInit()** and **PtlFini()** within a process (collection of threads) behave properly.

A child process does not inherit any portals resources from its parent. A child process must initialize Portals in order to obtain new, valid portals resources. If a child process fails to initialize Portals, behavior is undefined for both the parent and the child.

3.4.1 PtlInit

The **PtlInit()** function initializes the portals library. **PtlInit()** must be called at least once by a process before any thread makes a portals function call but may be safely called more than once. Each call to **PtlInit()** increments a reference count.

Function Prototype for PtlInit

```
int PtlInit (void);
```

Return Codes

PTL_OK	Indicates success.
PTL_FAIL	Indicates an error during initialization.

IMPLEMENTATION NOTE 12: Supporting fork()

If an implementation wants to support fork(), it must detect when **PtlInit()** is being called from a new process context and re-initialize the state of the Portals library.

3.4.2 PtlFini

The **PtlFini()** function allows an application to clean up after the portals library is no longer needed by a process. Each call to **PtlFini()** decrements the reference count that was incremented by **PtlInit()**. When the reference count reaches zero, all portals resources are freed. Once the portals resources are freed, calls to any of the functions defined by the portals API or use of the structures set up by the portals API will result in undefined behavior. Each call to **PtlInit()** should be matched by a corresponding **PtlFini()**.

Function Prototype for PtlFini

```
void PtlFini (void);
```

3.5 Network Interfaces

The portals API supports the use of multiple network interfaces. However, each interface is treated as an independent entity. Combining interfaces (e.g., “bonding” to create a higher bandwidth connection) must be implemented by the process or embedded in the underlying network. Interfaces are treated as independent entities to make it easier to cache information on individual network interface cards. In addition to supporting physical interfaces, each network interface can be initialized to provide either matching or non-matching portals addressing and either logical or physical addressing of network end-points through the data movement calls. These two options are independent (providing the full cross-product of possibilities) and must be provided for each physical interface such that a physical interface can be opened as four logical interfaces.

IMPLEMENTATION NOTE 13:	Logical network interfaces A logical interface is very similar to a physical interface. Like a physical interface, a logical interface is a “well known” interface — i.e. it is a specific physical interface with a specific set of properties. One additional burden placed on the implementation is the need for the initiator to place 2 bits in the message header to identify to the target the logical interface on which this message was sent. In addition, all logical interfaces associated with a single physical interface must share a single node ID and Portals process ID.

Once initialized, each logical interface provides a portal table and a collection of status registers. In order to facilitate the development of portable portals applications, a compliant implementation must provide at least 64 portal table entries. See Section 3.12 for a discussion of updating portal table entries using the **PtlMEAppend()** function. See Section 3.5.4 for a discussion of the **PtlNIStatus()** function, which can be used to read the value of a status register.

Every other type of portals object (e.g., memory descriptor, event queue, or match list entry) is associated with a specific logical network interface. The association to a logical network interface is established when the object is created and is encoded in the handle for the object.

Each logical network interface is initialized and shut down independently. The initialization routine, **PtlNIInit()**, returns a an interface object handle which is used in all subsequent portals operations. The **PtlNIFini()** function is used to shut down a logical interface and release any resources that are associated with the interface. Network interface handles are associated with processes, not threads. All threads in a process share all of the network interface handles.

> **Discussion**: *Proper initialization of a logical network interface that uses logical-end point addressing requires the user to pass in a requested mapping of logical* ranks *to physical node IDs and process IDs. To obtain this mapping, the process must first initialize a logical network interface that uses physical end-point addressing. The logical network interface that uses physical end-point addressing can be used to exchange a NID/PID map or the NID/PID map can be retrieved from a run-time system.*

The portals API also defines the **PtlNIStatus()** function (Section 3.5.4) to query the status registers for a logical network interface, and the **PtlNIHandle()** function (Section 3.5.5) to determine the logical network interface with which an object is associated.

3.5.1 The Network Interface Limits Type

The function **PtlNIInit()** accepts a pointer to a structure of desired limits and can fill a structure with the actual values supported by the network interface. The two lists are of type **ptl_ni_limits_t** and include the following members:

```
typedef struct {
    int max_mes;
    int max_mds;
    int max_cts;
    int max_eqs;
    int max_pt_index;
    int max_iovecs;
    int max_me_list;
    ptl_size_t max_msg_size;
    ptl_size_t max_atomic_size;
} ptl_ni_limits_t ;
```

Limits

max_mes	Maximum number of match list entries that can be allocated at any one time.
max_mds	Maximum number of memory descriptors that can be allocated at any one time.
max_eqs	Maximum number of event queues that can be allocated at any one time.
max_cts	Maximum number of counting events that can be allocated at any one time.
max_pt_index	Largest portal table index for this interface, valid indexes range from 0 to *max_pt_index*, inclusive. An interface must have a *max_pt_index* of at least 63.
max_iovecs	Maximum number of I/O vectors for a single memory descriptor for this interface.
max_me_list	Maximum number of match list entries that can be attached to any portal table index.
max_msg_size	Maximum size (in bytes) of a message (*put*, *get*, or *reply*).
max_atomic_size	Maximum size (in bytes) of an atomic operation.

3.5.2 PtlNIInit

The **PtlNIInit()** function initializes the portals API for a network interface (NI). A process using portals must call this function at least once before any other functions that apply to that interface. For subsequent calls to **PtlNIInit()** from within the same process (either by different threads or the same thread), the desired limits will be ignored and the call will return the existing network interface handle and the actual limits. Calls to **PtlNIInit()** increment a reference count on the network interface and must be matched by a call to **PtlNIFini()**.

Function Prototype for PtlNIInit

```
int  PtlNIInit (ptl_interface_t      iface ,
                unsigned int         options ,
                ptl_pid_t            pid,
                ptl_ni_limits_t      *desired ,
                ptl_ni_limits_t      *actual ,
                ptl_size_t           map_size,
                ptl_process_id_t     *desired_mapping,
                ptl_process_id_t     *actual_mapping,
                ptl_handle_ni_t      *ni_handle );
```

Arguments

iface	**input**	Identifies the network interface to be initialized. (See Section 3.2.5 for a discussion of values used to identify network interfaces.)
options	**input**	This field contains options that are requested for the network interface. Values for this argument can be constructed using a bitwise OR of the values defined below. Either PTL_NI_MATCHING or PTL_NI_NO_MATCHING must be set, but not both. Either PTL_NI_LOGICAL or PTL_NI_PHYSICAL must be set, but not both.
pid	**input**	Identifies the desired process identifier (for well known process identifiers). The value PTL_PID_ANY may be used to let the portals library select a process identifier.
desired	**input**	If not NULL, points to a structure that holds the desired limits.
actual	**output**	If not NULL, on successful return, the location pointed to by actual will hold the actual limits.
map_size	**input**	Contains the size of the map being passed in (zero for NULL). This field is ignored if the PTL_NI_LOGICAL option is not set.
desired_mapping	**input**	If not NULL, points to an array of structures that holds the desired mapping of logical indentifiers to NID/PID pairs. This field is ignored if the PTL_NI_LOGICAL option is not set.
actual_mapping	**output**	If the PTL_NI_LOGICAL option is set, on successful return, the location pointed to by *actual_mapping* will hold the actual mapping of logical identifiers to NID/PID pairs.
ni_handle	**output**	On successful return, this location will hold a the interface handle.

options

PTL_NI_MATCHING	Request that the interface specified in *iface* be opened with matching enabled.
PTL_NI_NO_MATCHING	Request that the interface specified in *iface* be opened with matching disabled. PTL_NI_MATCHING and PTL_NI_NO_MATCHING are mutually exclusive.
PTL_NI_LOGICAL	Request that the interface specified in *iface* be opened with logical end-point addressing (e.g. MPI communicator and rank or SHMEM PE).
PTL_NI_PHYSICAL	Request that the interface specified in *iface* be opened with physical end-point addressing (e.g. NID/PID). PTL_NI_LOGICAL and PTL_NI_PHYSICAL are mutually exclusive.

Discussion: *The use of desired is implementation dependent. In particular, an implementation may choose to ignore this argument*

Discussion: *Each interface has its own sets of limits. In implementations that support multiple interfaces, the limits passed to and returned by* ***PtlNIInit()*** *apply only to the interface specified in iface.*

The desired limits are used to offer a hint to an implementation as to the amount of resources needed, and the implementation returns the actual limits available for use. In the case where an implementation does not have any pre-defined limits, it is free to return the largest possible value permitted by the corresponding type (e.g., INT_MAX). A quality implementation will enforce the limits that are returned and take the appropriate action when limits are exceeded, such as using the **PTL_NO_SPACE** return code. The caller is permitted to use maximum values for the desired fields to indicate that the limit should be determined by the implementation.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_IFACE_INVALID	Indicates that *iface* is not a valid network interface.
PTL_PID_INVALID	Indicates that *pid* is not a valid process identifier.
PTL_PID_INUSE	Indicates that *pid* is currently in use.
PTL_SEGV	Indicates that *actual* or *ni_handle* is not NULL or a legal address, or that *desired* is not NULL and does not point to a valid address.

IMPLEMENTATION NOTE 14:

Multiple calls to **PtlNIInit()**

If **PtlNIInit()** gets called more than once *per logical interface*, then the implementation should fill in *actual*, *actual_mapping* and *ni_handle*. It should ignore *pid*. **PtlGetId()** (Section 3.8) can be used to retrieve the *pid*.

3.5.3 PtlNIFini

The **PtlNIFini()** function is used to release the resources allocated for a network interface. The release of network interface resources is based on a reference count that is incremented by **PtlNIInit()** and decremented by **PtlNIFini()**. Resources can only be released when the reference count reaches zero. Once the release of resources has begun, the results of pending API operations (e.g., operations initiated by another thread) for this interface are undefined. Similarly, the effects of incoming operations (*put*, *get*, *atomic*) or return values (*acknowledgment* and *reply*) for this interface are undefined.

Function Prototype for PtlNIFini

```
int  PtlNIFini (ptl_handle_ni_t    ni_handle );
```

Arguments

ni_handle	**input**	An interface handle to shut down.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.

3.5.4 PtlNIStatus

The **PtlNIStatus()** function returns the value of a status register for the specified interface. (See Section 3.2.7 for more information on status register indexes and status register values.)

Function Prototype for PtlNIStatus

```
int  PtlNIStatus (ptl_handle_ni_t    ni_handle ,
                  ptl_sr_index_t     status_register  ,
                  ptl_sr_value_t     *status );
```

Arguments

ni_handle	**input**	An interface handle
status_register	**input**	The index of the status register
status	**output**	On successful return, this location will hold the current value of the status register.

Discussion: *Only two status registers are currently required: a drop count register (`PTL_SR_DROP_COUNT`) and an attempted permissions violation register (`PTL_SR_PERMISSIONS_VIOLATIONS`). Implementations may define additional status registers.*

Identifiers for the indexes associated with these registers should start with the prefix PTL_SR_.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_SR_INDEX_INVALID	Indicates that *status_register* is not a valid status register.
PTL_SEGV	Indicates that *status* is not a legal address.

3.5.5 PtlNIHandle

The **PtlNIHandle()** function returns the network interface handle with which the object identified by *handle* is associated. If the object identified by *handle* is a network interface, this function returns the same value it is passed.

Function Prototype for PtlNIHandle

```
int PtlNIHandle(ptl_handle_any_t    handle,
                ptl_handle_ni_t     *ni_handle);
```

Arguments

handle	**input**	The object handle.
ni_handle	**output**	On successful return, this location will hold the network interface handle associated with *handle*.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_HANDLE_INVALID	Indicates that *handle* is not a valid handle.
PTL_SEGV	Indicates that *ni_handle* is not a legal address.

IMPLEMENTATION NOTE 15: Object encoding in handle

Every handle should encode the network interface and the object identifier relative to this handle.

3.6 Portal Table Entries

A portal index refers to a portal table entry. The assignment of these indexes can either be statically or dynamically managed, and will typically be a combination of both. A portal table entry must be allocated before being used.

3.6.1 PtlPTAlloc

The **PtlPTAlloc()** function allocates a portal table entry and sets flags that pass options to the implementation.

Function Prototype for PtlPTAlloc

```
int  PtlPTAlloc(ptl_handle_ni_t   ni_handle,
                unsigned int      options,
                ptl_handle_eq_t   eq_handle;
                ptl_pt_index_t    pt_index_req ,
                ptl_pt_index_t    *pt_index );
```

Arguments

ni_handle	**input**	The interface handle to use.
options	**input**	This field contains options that are requested for the portal index. Values for this argument can be constructed using a bitwise OR of the values defined below.
eq_handle	**input**	The event queue handle used to log the operations performed on match list entries attached to the portal table entry. The *eq_handle* attached to a portal table entry must refer to an event queue containing **ptl_target_event_t** type events. If this argument is PTL_EQ_NONE, operations performed on this portal table entry are not logged.
pt_index_req	**input**	The value of the portal index that is requested. If the value is set to PTL_PT_ANY, the implementation can return any portal index.
pt_index	**output**	On successful return, this location will hold the portal index that has been allocated.

options

PTL_PT_ONLY_USE_ONCE	Hint to the underlying implementation that all entries attached to this portal table entry will have the PTL_ME_USE_ONCE or PTL_LE_USE_ONCE option set.
PTL_PT_FLOW_CONTROL	Enable flow control on this portal table entry (see Section 2.3).

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates that *iface* is not a valid network interface handle.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.

PTL_PT_FULL	Indicates that there are no free entries in the portal table.
PTL_PT_IN_USE	Indicates that the Portal table entry requested is in use.
PTL_PT_EQ_NEEDED	Indicates that flow control is enabled and there is no EQ attached.

3.6.2 PtlPTFree

The **PtlPTFree()** function releases the resources associated with a portal table entry.

Function Prototype for PtlPTFree

```
int  PtlPTFree( ptl_handle_ni_t    ni_handle ,
                ptl_pt_index_t     pt_index );
```

Arguments

ni_handle	**input**	The interface handle on which the *pt_index* should be freed.
pt_index	**input**	The portal index that is to be freed.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_PT_INDEX_INVALID	Indicates that *pt_index* is not a valid portal table index.
PTL_PT_IN_USE	Indicates that *pt_index* is currently in use (e.g. a match list entry is still attached).
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.

3.6.3 PtlPTDisable

The **PtlPTDisable()** function indicates to an implementation that no new messages should be accepted on that portal table entry. The function blocks until the portal table entry status has been updated, all messages being actively processed are completed, and all events are posted.

Function Prototype for PtlPTDisable

```
int  PtlPTDisable(ptl_handle_ni_t    ni_handle ,
                  ptl_pt_index_t     pt_index );
```

Arguments

ni_handle	**input**	The interface handle to use.
pt_index	**input**	The portal index that is to be disable.

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates that *iface* is not a valid network interface handle.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.

Discussion: *After successful completion, no other messages will be accepted on this portal table entry and no more events associated with this portal table entry will be delivered. Replies arriving at this initiator will continue to succeed.*

3.6.4 PtlPTEnable

The **PtlPTEnable()** function indicates to an implementation that a previously disabled portal table entry should be re-enabled. This is used to enable portal table entries that were automatically or manually disabled. The function blocks until the portal table entry status has been updated.

Function Prototype for PtlPTEnable

```
int  PtlPTEnable(ptl_handle_ni_t   ni_handle ,
                 ptl_pt_index_t    pt_index );
```

Arguments

ni_handle	**input**	The interface handle to use.
pt_index	**input**	The value of the portal index to enable.

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates that *iface* is not a valid network interface handle.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.

3.7 User Identification

Every process runs on behalf of a user. User identifiers travel in the trusted portion of the header of a portals message. They can be used at the *target* to limit access via access controls (Section 3.11 and Section 3.12).

3.7.1 PtlGetUid

The **PtlGetUid()** function is used to retrieve the user identifier of a process.

Function Prototype for PtlGetUid

```
int  PtlGetUid(ptl_handle_ni_t    ni_handle ,
               ptl_uid_t          *uid);
```

Arguments

ni_handle	**input**	A network interface handle.
uid	**output**	On successful return, this location will hold the user identifier for the calling process.

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_SEGV	Indicates that *uid* is not a legal address.

3.8 Process Identification

Processes that use the portals API can be identified using a node identifier and process identifier. Every node accessible through a network interface has a unique node identifier and every process running on a node has a unique process identifier. As such, any process in the computing system can be uniquely identified by its node identifier and process identifier. The node identifier and process identifier can be aggregated by the application into a rank, which is translated by the implementation into a network identifier and process identifier.

The portals API defines a type, **ptl_process_id_t**, for representing process identifiers, and a function, **PtlGetId()**, which can be used to obtain the identifier of the current process.

> **Discussion**: *The portals API does not include thread identifiers. Messages are delivered to processes (address spaces) not threads (contexts of execution).*

3.8.1 The Process Identification Type

The **ptl_process_id_t** type is a union that can represent the a node as either a physical address or a logical address within the machine. The physical address uses two identifiers to represent a process identifier: a node identifier *nid* and a process identifier *pid*. In turn, a logical address uses a logical index within a translation table specified by the application (the *rank*) to identify another process.

```
typedef union    {
    struct  {
        ptl_nid_t   nid;
        ptl_pid_t   pid;
    } phys;
    ptl_rank_t  rank;
} ptl_process_id_t  ;
```

3.8.2 PtlGetId

Function Prototype for PtlGetId

```
int  PtlGetId (ptl_handle_ni_t      ni_handle ,
               ptl_process_id_t     *id);
```

Arguments

ni_handle	**input**	A network interface handle.
id	**output**	On successful return, this location will hold the identifier for the calling process.

Discussion: *Note that process identifiers and ranks are dependent on the network interface(s). In particular, if a node has multiple interfaces, it may have multiple process identifiers and multiple ranks.*

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_SEGV	Indicates that *id* is not a legal address.

3.9 Process Aggregation

It is useful in the context of a parallel machine to represent all of the processes in a parallel job through an aggregate identifier. The portals API provides a mechanism for supporting such job identifiers for these systems. In order to be fully supported, job identifiers must be included as a trusted part of a message header.

The job identifier is an opaque identifier shared between all of the distributed processes of an application running on a parallel machine. All application processes and job-specific support programs, such as the parallel job launcher, share the same job identifier. This identifier is assigned by the runtime system upon job launch and is guaranteed to be unique among application jobs currently running on the entire distributed system. An individual serial process may be assigned a job identifier that is not shared with any other processes in the system or can be assigned the constant `PTL_JID_NONE`.

3.9.1 PtlGetJid

Function Prototype for PtlGetJid

```
int  PtlGetJid (ptl_handle_ni_t    ni_handle ,
                ptl_jid_t          *jid );
```

Arguments

ni_handle	**input**	A network interface handle.
jid	**output**	On successful return, this location will hold the job identifier for the calling process. `PTL_JID_NONE` may be returned for a serial job, if a job identifier is not assigned.

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates the *ni_handle* is not a valid network interface handle.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_SEGV	Indicates that *jid* is not a legal address.

Discussion: *The notion of a job identifier is fairly closely tied to a run-time system. It is expected that the run-time system will set this value. For implementations without a run-time system, `PTL_JID_NONE` may be assigned. It would probably be a bad idea to use job ID on those systems for access control.*

3.10 Memory Descriptors

A memory descriptor contains information about a region of a process' memory and optionally points to an event queue where information about the operations performed on the memory descriptor are recorded. Memory descriptors are initiator side resources that are used to encapsulate an association with a network interface (NI) with a description of a memory region. They provide an interface to register memory (for operating systems that require it) and to carry that information across multiple operations (an MD is persistent until released). **PtlMDBind()** is used to create a memory descriptor and **PtlMDRelease()** is used to unlink and release the resources associated with a memory descriptor.

3.10.1 The Memory Descriptor Type

The **ptl_md_t** type defines the visible parts of a memory descriptor. Values of this type are used to initialize the memory descriptors.

```
typedef struct {
    void            *start ;
    ptl_size_t      length ;
    unsigned int    options ;
    ptl_handle_eq_t eq_handle;
    ptl_handle_ct_t ct_handle ;
} ptl_md_t;
```

Members

start, *length*
: Specify the memory region associated with the memory descriptor. The *start* member specifies the starting address for the memory region and the *length* member specifies the length of the region. There are no alignment restrictions on the starting address or the length of the region; although unaligned messages may be slower (i.e., lower bandwidth and/or longer latency) on some implementations.

options
: Specifies the behavior of the memory descriptor. Options include the use of scatter/gather vectors and disabling of end events associated with this memory descriptor. Values for this argument can be constructed using a bitwise OR of the following values:

`PTL_MD_EVENT_DISABLE`
: Specifies that this memory descriptor should not generate events.

`PTL_MD_EVENT_SUCCESS_DISABLE`
: Specifies that this memory descriptor should not generate events that indicate success. This is useful in scenarios where the application does not need normal events, but does require failure information to enhance reliability.

`PTL_MD_EVENT_CT_SEND`
: Enable the counting of `PTL_EVENT_SEND` events.

`PTL_MD_EVENT_CT_REPLY`
: Enable the counting of `PTL_EVENT_REPLY` events.

`PTL_MD_EVENT_CT_ACK`
: Enable the counting of `PTL_EVENT_ACK` events.

`PTL_MD_UNORDERED`
: Indicate to the Portals implementation that messages sent from this memory descriptor do not have to arrive at the target in order.

`PTL_MD_REMOTE_FAILURE_DISABLE`
: Indicate to the Portals implementation that failures requiring notification from the target should not be delivered to the local application. This prevents the local events (e.g. `PTL_EVENT_SEND`) from having to wait for a round-trip notification before delivery.

`PTL_IOVEC`
: Specifies that the `start` argument is a pointer to an array of type **ptl_iovec_t** (Section 3.10.2) and the `length` argument is the length of the array of **ptl_iovec_t** elements. This allows for a scatter/gather capability for memory descriptors. A scatter/gather memory descriptor behaves exactly as a memory descriptor that describes a single virtually contiguous region of memory.

eq_handle
: The event queue handle used to log the operations performed on the memory region. If this argument is `PTL_EQ_NONE`, operations performed on this memory descriptor are not logged.

ct_handle
: A handle for counting type events associated with the memory region. If this argument is `PTL_CT_NONE`, operations performed on this memory descriptor are not counted.

3.10.2 The I/O Vector Type

The **ptl_iovec_t** type is used to describe scatter/gather buffers of a match list entry or memory descriptor in conjunction with the PTL_IOVEC option. The **ptl_iovec_t** type is intended to be a type definition of the `struct iovec` type on systems that already support this type.

```
typedef struct {
    void          *iov_base;
    ptl_size_t    iov_len;
} ptl_iovec_t;
```

Members

iov_base	The byte aligned start address of the vector element
iov_len	The length (in bytes) of the vector element

Discussion: *Performance conscious users should not mix offsets (local or remote) with* ***ptl_iovec_t*** *. While this is a* supported *operation, it is unlikely to perform well in most implementations.*

IMPLEMENTATION NOTE 16:

Support of I/O Vector Type and Offset

The implementation is required to support the mixing of the **ptl_iovec_t** type with offsets (local and remote); however, it will be difficult to make this perform well in the general case. The correct behavior in this scenario is to treat the region described by the **ptl_iovec_t** type as if it were a single contiguous region. In some cases, this may require walking the entire scatter/gather list to find the correct location for depositing the data.

3.10.3 PtlMDBind

The **PtlMDBind()** operation is used to create a memory descriptor to be used by the *initiator*. On systems that require memory registration, the **PtlMDBind()** operation would invoke the appropriate memory registration functions.

Function Prototype for PtlMDBind

```
int PtlMDBind(ptl_handle_ni_t     ni_handle,
              ptl_md_t            md,
              ptl_handle_md_t     *md_handle);
```

Arguments

ni_handle	**input**	The network interface handle with which the memory descriptor will be associated.

md	**input**	Provides initial values for the user-visible parts of a memory descriptor. Other than its use for initialization, there is no linkage between this structure and the memory descriptor maintained by the API.
md_handle	**output**	On successful return, this location will hold the newly created memory descriptor handle. The *md_handle* argument must be a valid address and cannot be NULL.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_MD_ILLEGAL	Indicates that *md* is not a legal memory descriptor. This may happen because the memory region defined in *md* is invalid or because the network interface associated with the *eq_handle* or the *ct_handle* in *md* is not the same as the network interface, *ni_handle*.
PTL_EQ_INVALID	Indicates that the event queue associated with *md* is not valid.
PTL_CT_INVALID	Indicates that the counting event associated with *md* is not valid.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the memory descriptor.
PTL_SEGV	Indicates that *md_handle* is not a legal address.

3.10.4 PtlMDRelease

The **PtlMDRelease()** function releases the internal resources associated with a memory descriptor. (This function does not free the memory region associated with the memory descriptor; i.e., the memory the user allocated for this memory descriptor.) Only memory descriptors with no pending operations may be unlinked.

IMPLEMENTATION NOTE 17: Unique memory descriptor handles

An implementation will be greatly simplified if the encoding of memory descriptor handles does not get reused. This makes debugging easier, and it avoids race conditions between threads calling **PtlMDRelease()** and **PtlMDBind()**.

Function Prototype for PtlMDRelease

```
int PtlMDRelease(ptl_handle_md_t md_handle);
```

Arguments

md_handle	**input**	The memory descriptor handle to be released.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *md_handle* is not a valid memory descriptor handle.
PTL_MD_IN_USE	Indicates that *md_handle* has pending operations and cannot be released. See Figure 3.1 for when data structures are considered to be in use.

3.11 List Entries and Lists

A list is a chain of list entries. Examples of lists include the priority list and the overflow list. Each list entry (LE) describes a memory region and includes a set of options. It is the target side analogue of the memory descriptor (MD). A list is created using the **PtlLEAppend()** function, which appends a single list entry to the specified list on the specified portal index, and returns the list entry handle. List entries can be dynamically removed from a list using the **PtlLEUnlink()** function.

List entries can be appended to either the priority list or the overflow list associated with a portal table entry; however, when attached to an overflow list, additional semantics are implied that require the implementation to track messages that arrive in list entries. Essentially, the memory region identified is simply provided to the implementation for use in managing unexpected messages. Buffers provided in the overflow list will post an event (`PTL_EVENT_UNLINK`) when the buffer space has been consumed, to notify the application that more buffer space may be needed. When the application is free to reuse the buffer (i.e. the implementation is done with it), another event (`PTL_EVENT_FREE`) will be posted. A third type of event (`PTL_EVENT_DROPPED`) will be posted if a message arrives, finds no entries the priority list, and the overflow list is exhausted.

> **Discussion**: *It is the responsibility of the application to ensure that the implementation has sufficient buffer space to manage unexpected messages. Failure to do will cause messages to be dropped and an `PTL_EVENT_DROPPED` to be posted. Note that overflow events can readily exhaust the event queue. Proper use of the API will generally require the application to post at least two (and typically several) buffers so that the application has time to notice the `PTL_EVENT_UNLINK` and replace the buffer. In many usage scenarios, however, the application may choose to have only persistent list entries in the priority list. Thus, overflow list entries will not be required.*

> **Discussion**: *It is the responsibility of the implementation to determine when a buffer unlinked from an overflow list can be reused. It must note that it is no longer holding state in the buffer and post a `PTL_EVENT_FREE` event.*

List entries can be appended to a network interface with the `PTL_NI_NO_MATCHING` option set (a non-matching network interface). A matching network interface requires a match list entry.

3.11.1 The List Entry Type

The **ptl_le_t** type defines the visible parts of a list entry. Values of this type are used to initialize the list entries.

> **Discussion**: *The list entry (LE) has a number of fields in common with the memory descriptor (MD). The overlapping fields have the same meaning in the LE as in the MD; however, since initiator and target resources are decoupled, the MD is not a proper subset of the LE, and the options field has different meaning based on whether it is used at an initiator or target, it was deemed undesirable and cumbersome to include a "target MD" structure that would be included as an entry in the LE.*

Discussion: *The default behavior from Portals 3.3 (no truncation and locally managed offsets) has been changed to match the default semantics of the list entry, which does not provide matching.*

To facilitate acccess control to both list entries and match list entries, the **ptl_ac_id_t** is defined as a union of a job ID and a user ID. A **ptl_ac_id_t** is attached to each list entry or match list entry to control which user (or which job, as selected by an option) can access the entry. Either field can specify a wildcard.

```
typedef union {
    ptl_jid_t       jid ;
    ptl_uid_t       uid;
} ptl_ac_id_t ;
```

Members

uid — The user identifier of the *initiator* that may access the associated list entry or match list entry. This may be set to PTL_UID_ANY to allow access by any user.

jid — The job identifier of the *initiator* that may access the associated list entry or match list entry. This may be set to PTL_JID_ANY to allow access by any job.

```
typedef struct {
    void            *start ;
    ptl_size_t      length ;
    ptl_handle_ct_t ct_handle ;
    ptl_ac_id_t     ac_id ;
    unsigned int    options ;
} ptl_le_t ;
```

Members

start, *length* — Specify the memory region associated with the match list entry. The *start* member specifies the starting address for the memory region and the *length* member specifies the length of the region. The *start* member can be NULL provided that the *length* member is zero. Zero-length buffers (NULL LE) are useful to record events. There are no alignment restrictions on buffer alignment, the starting address or the length of the region; although messages that are not natively aligned (e.g. to a four byte or eight byte boundary) may be slower (i.e., lower bandwidth and/or longer latency) on some implementations.

ct_handle — A handle for counting type events associated with the memory region. If this argument is PTL_CT_NONE, operations performed on this list entry are not counted.

ac_id

Specifies either the user ID or job ID (as selected by the options) that may access this list entry. Either the user ID or job ID may be set to a wildcard (PTL_UID_ANY or PTL_JID_ANY). If the access control check fails, then the message is dropped without modifying Portals state. This is treated as a permissions failure and the **PtlNIStatus()** register indexed by PTL_SR_PERMISSIONS_VIOLATIONS is incremented. This failure is also indicated to the initiator through the *ni_fail_type* in the PTL_EVENT_SEND event, unless the PTL_MD_REMOTE_FAILURE_DISABLE option is set.

options

Specifies the behavior of the list entry. The following options can be selected: enable *put* operations (yes or no), enable *get* operations (yes or no), offset management (local or remote), message truncation (yes or no), acknowledgment (yes or no), use scatter/gather vectors and disable events. Values for this argument can be constructed using a bitwise OR of the following values:

PTL_LE_OP_PUT

Specifies that the list entry will respond to *put* operations. By default, list entries reject *put* operations. If a *put* operation targets a list entry where PTL_LE_OP_PUT is not set, it is treated as a permissions failure.

PTL_LE_OP_GET

Specifies that the list entry will respond to *get* operations. By default, list entries reject *get* operations. If a *get* operation targets a list entry where PTL_LE_OP_GET is not set, it is treated as a permissions failure

Note: It is not considered an error to have a list entry that does not respond to either *put* or *get* operations: Every list entry responds to *reply* operations. Nor is it considered an error to have a list entry that responds to both *put* and *get* operations. In fact, it is often desirable for a list entry used in an *atomic* operation to be configured to respond to both *put* and *get* operations.

PTL_LE_USE_ONCE

Specifies that the list entry will only be used once and then unlinked. If this option is not set, the list entry persists until it is explicitly unlinked is triggered.

PTL_LE_ACK_DISABLE

Specifies that an *acknowledgment* should *not* be sent for incoming *put* operations, even if requested. By default, acknowledgments are sent for *put* operations that request an acknowledgment. This applies to both standard and counting type events. Acknowledgments are never sent for *get* operations. The data sent in the *reply* serves as an implicit acknowledgment.

PTL_IOVEC

Specifies that the start argument is a pointer to an array of type **ptl_iovec_t** (Section 3.10.2) and the length argument is the length of the array. This allows for a scatter/gather capability for list entries. A scatter/gather list entry behaves exactly as a list entry that describes a single virtually contiguous region of memory. All other semantics are identical.

PTL_LE_EVENT_DISABLE

Specifies that this list entry should not generate events.

PTL_LE_EVENT_SUCCESS_DISABLE

Specifies that this list entry should not generate events that indicate success. This is useful in scenarios where the application does not need normal events, but does require failure information to enhance reliability.

PTL_LE_EVENT_OVER_DISABLE

Specifies that this list entry should not generate overflow list events.

PTL_LE_EVENT_UNLINK_DISABLE

Specifies that this list entry should not generate unlink (PTL_EVENT_UNLINK) or free (PTL_EVENT_FREE) events.

PTL_LE_EVENT_CT_GET

Enable the counting of PTL_EVENT_GET events.

PTL_LE_EVENT_CT_PUT	Enable the counting of PTL_EVENT_PUT events.
PTL_LE_EVENT_CT_PUT_OVERFLOW	Enable the counting of PTL_EVENT_PUT_OVERFLOW events.
PTL_LE_EVENT_CT_ATOMIC	Enable the counting of PTL_EVENT_ATOMIC events.
PTL_LE_EVENT_CT_ATOMIC_OVERFLOW	Enable the counting of PTL_EVENT_ATOMIC_OVERFLOW events.
PTL_LE_AUTH_USE_JID	Use job ID for authentication instead of user ID. By default, the user ID must match to allow a message to access a list entry.

3.11.2 PtlLEAppend

The **PtlLEAppend()** function creates a single list entry and appends this entry to the end of the list specified by *ptl_list* associated with the portal table entry specified by *pt_index* for the portal table for *ni_handle*. If the list is currently uninitialized, the **PtlLEAppend()** function creates the first entry in the list.

When a list entry is posted to a list, the overflow list is checked to see if a message has arrived prior to posting the list entry. If so, a PTL_EVENT_PUT_OVERFLOW event is generated. No searching is performed when a list entry is posted to the overflow list.

```
typedef enum {
    PTL_PRIORITY_LIST, PTL_OVERFLOW, PTL_PROBE_ONLY
} ptl_list_t ;
```

LE List Types

PTL_PRIORITY_LIST	The priority list associated with a portal table entry
PTL_OVERFLOW	The overflow list associated with a portal table entry
PTL_PROBE_ONLY	Do not attach to a list. Use the LE to probe the overflow list, without consuming an item in the list and without being attached anywhere.

Function Prototype for PtlLEAppend

```
int PtlLEAppend(ptl_handle_ni_t    ni_handle ,
                ptl_pt_index_t     pt_index ,
                ptl_le_t           le ,
                ptl_list_t         ptl_list ,
                void               *user_ptr ,
                ptl_handle_le_t    *le_handle );
```

Arguments

ni_handle	**input**	The interface handle to use.
pt_index	**input**	The portal table index where the list entry should be appended.

le	**input**	Provides initial values for the user-visible parts of a list entry. Other than its use for initialization, there is no linkage between this structure and the list entry maintained by the API.
ptl_list	**input**	Determines whether the list entry is appended to the priority list, appended to the overflow list, or simply queries the overflow list.
user_ptr	**input**	A user-specified value that is associated with each command that can generate an event. The value does not need to be a pointer, but must fit in the space used by a pointer. This value (along with other values) is recorded in events associated with operations on this list entry[1].
le_handle	**output**	On successful return, this location will hold the newly created list entry handle.

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_PT_INDEX_INVALID	Indicates that *pt_index* is not a valid portal table index.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the match list entry.
PTL_LE_LIST_TOO_LONG	Indicates that the resulting list is too long. The maximum length for a list is defined by the interface.

3.11.3 PtlLEUnlink

The **PtlLEUnlink()** function can be used to unlink a list entry from a list. This operation also releases any resources associated with the list entry. It is an error to use the list entry handle after calling **PtlLEUnlink()**.

Function Prototype for PtlLEUnlink

```
int PtlLEUnlink(ptl_handle_le_t le_handle );
```

Arguments

le_handle	**input**	The list entry handle to be unlinked.

Discussion: *If this list entry has pending operations; e.g., an unfinished reply operation, then **PtlLEUnlink()** will return **PTL_LE_IN_USE**, and the list entry will not be unlinked. This essentially creates a race between the application retrying the unlink operation and a new operation arriving. This is believed to be reasonable as the application rarely wants to unlink an LE while new operations are arriving to it.*

[1] Tying commands to a user-defined value is useful at the target when the command needs to be associated with a data structure maintained by the process outside of the portals library. For example, an MPI implementation can set the *user_ptr* argument to the value of an MPI Request. This direct association allows for processing of list entries by the MPI implementation without a table lookup or a search for the appropriate MPI Request.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_LE_INVALID	Indicates that *le_handle* is not a valid list entry handle.
PTL_LE_IN_USE	Indicates that the list entry has pending operations and cannot be unlinked.

3.12 Match List Entries and Matching Lists

Matching list entries add matching semantics to the basic list constructs. Each match list entry (ME) adds a set of match criteria to the basic memory region description in the list entry. The match criteria added can be used to reject incoming requests based on process identifier or the match bits provided in the request. A match list (priority list or overflow list) is created using the **PtlMEAppend()** function, which appends a single match list entry to the specified portal index, and returns the match list entry handle. Matching list entries can be dynamically removed from a list using the **PtlMEUnlink()** function.

Matching list entries can be appended to either the priority list or the overflow list associated with a portal table entry; however, when attached to an overflow list, additional semantics are implied that require the implementation to track messages that arrive in match list entries. Essentially, the memory region identified is simply provided to the implementation for use in managing unexpected messages; however, the application may use the match bits and other matching criteria to further constrain how these buffers are used. Buffers provided in the overflow list will post an event (`PTL_EVENT_UNLINK`) when the buffer space has been consumed, to notify the application that more buffer space may be needed. When the application is free to reuse the buffer (i.e. the implementation is done with it), another event (`PTL_EVENT_FREE`) will be posted. A third type of event (`PTL_EVENT_DROPPED`) will be posted if a message arrives, does not match in the priority list, and the overflow list is exhausted.

> **Discussion**: *It is the responsibility of the application to ensure that the implementation has sufficient buffer space to manage unexpected messages. Failure to do will cause messages to be dropped and an `PTL_EVENT_DROPPED` to be posted. Note that overflow events can readily exhaust the event queue. Proper use of the API will generally require the application to post at least two (and typically several) buffers so that the application has time to notice the `PTL_EVENT_UNLINK` and replace the buffer.*

> **Discussion**: *It is the responsibility of the implementation to determine when a buffer unlinked from an overflow list can be reused. It must note that it is no longer holding state in the buffer and post a `PTL_EVENT_FREE` event.*

Matching list entries can be appended to a network interface without the `PTL_NI_NO_MATCHING` option set; however, an NI with the `PTL_NI_LOGICAL` option set changes the interpretation of the *match_id*.

3.12.1 The Match List Entry Type

The **ptl_me_t** type defines the visible parts of a match list entry. Values of this type are used to initialize and update the match list entries.

> **Discussion**: *The match list entry (ME) has a number of fields in common with the memory descriptor (MD). The overlapping fields have the same meaning in the ME as in the MD; however, since initiator and target resources are decoupled, the MD is not a proper subset of the ME, and the options field has*

different meaning based on whether it is used at an initiator or target, it was deemed undesirable and cumbersome to include a "target MD" structure that would be included as an entry in the ME.

```
typedef struct {
    void                *start ;
    ptl_size_t          length ;
    ptl_handle_ct_t     ct_handle ;
    ptl_size_t          min_free ;
    ptl_ac_id_t         ac_id ;
    unsigned int        options ;
    ptl_process_id_t    match_id;
    ptl_match_bits_t    match_bits ;
    ptl_match_bits_t    ignore_bits  ;
} ptl_me_t;
```

Members

start, *length*
: Specify the memory region associated with the match list entry. The *start* member specifies the starting address for the memory region and the *length* member specifies the length of the region. The *start* member can be NULL provided that the *length* member is zero. Zero-length buffers (NULL ME) are useful to record events. There are no alignment restrictions on buffer alignment, the starting address or the length of the region; although unaligned messages may be slower (i.e., lower bandwidth and/or longer latency) on some implementations.

ct_handle
: A handle for counting type events associated with the memory region. If this argument is `PTL_CT_NONE`, operations performed on this match list entry are not counted.

min_free
: When the unused portion of a match list entry (length - local offset) falls below this value, the match list entry automatically unlinks . This value is only used if the `PTL_ME_MIN_FREE` option is specified and `PTL_ME_MANAGE_LOCAL` is set.

ac_id
: Specifies either the user ID or job ID (as selected by the options) that may access this match list entry. Either the user ID or job ID may be set to a wildcard (`PTL_UID_ANY` or `PTL_JID_ANY`). If the access control check fails, then the message is dropped without modifying Portals state. This is treated as a permissions failure and the **PtlNIStatus()** register indexed by `PTL_SR_PERMISSIONS_VIOLATIONS` is incremented. This failure is also indicated to the initiator through the *ni_fail_type* in the `PTL_EVENT_SEND` event, unless the `PTL_MD_REMOTE_FAILURE_DISABLE` option is set.

options
: Specifies the behavior of the match list entry. The following options can be selected: enable *put* operations (yes or no), enable *get* operations (yes or no), offset management (local or remote), message truncation (yes or no), acknowledgment (yes or no), use scatter/gather vectors and disable events. Values for this argument can be constructed using a bitwise OR of the following values:

PTL_ME_OP_PUT	Specifies that the match list entry will respond to *put* operations. By default, match list entries reject *put* operations. If a *put* operation targets a list entry where PTL_ME_OP_PUT is not set, it is treated as a permissions failure.
PTL_ME_OP_GET	Specifies that the match list entry will respond to *get* operations. By default, match list entries reject *get* operations. If a *get* operation targets a list entry where PTL_ME_OP_GET is not set, it is treated as a permissions failure. **Note**: It is not considered an error to have a match list entry that does not respond to either *put* or *get* operations: Every match list entry responds to *reply* operations. Nor is it considered an error to have a match list entry that responds to both *put* and *get* operations. In fact, it is often desirable for a match list entry used in an *atomic* operation to be configured to respond to both *put* and *get* operations.
PTL_ME_MANAGE_LOCAL	Specifies that the offset used in accessing the memory region is managed locally. By default, the offset is in the incoming message. When the offset is maintained locally, the offset is incremented by the length of the request so that the next operation (*put* and/or *get*) will access the next part of the memory region. Note that only one offset variable exists per match list entry. If both *put* and *get* operations are performed on a match list entry, the value of that single variable is updated each time.
PTL_ME_NO_TRUNCATE	Specifies that the length provided in the incoming request cannot be reduced to match the memory available in the region. This can cause the match to fail. (The memory available in a memory region is determined by subtracting the offset from the length of the memory region.) By default, if the length in the incoming operation is greater than the amount of memory available, the operation is truncated.
PTL_ME_USE_ONCE	Specifies that the match list entry will only be used once and then unlinked. If this option is not set, the match list entry persists until another unlink condition is triggered.
PTL_ME_MAY_ALIGN	Indicate that messages deposited into this match list entry may be aligned by the implementation to a performance optimizing boundary. Essentially, this is a performance hint to the implementation to indicate that the application does not care about the specific placement of the data. This option is only relevant when the PTL_ME_MANAGE_LOCAL option is set.
PTL_ME_ACK_DISABLE	Specifies that an *acknowledgment* should *not* be sent for incoming *put* operations, even if requested. By default, acknowledgments are sent for *put* operations that request an acknowledgment. This applies to both standard and counting type events. Acknowledgments are never sent for *get* operations. The data sent in the *reply* serves as an implicit acknowledgment.
PTL_IOVEC	Specifies that the `start` argument is a pointer to an array of type **ptl_iovec_t** (Section 3.10.2) and the `length` argument is the length of the array. This allows for a scatter/gather capability for match list entries. A scatter/gather match list entry behaves exactly as a match list entry that describes a single virtually contiguous region of memory. All other semantics are identical.
PTL_ME_MIN_FREE	Specifies that the *min_free* field in the match list entry is to be used. This option is only used if PTL_ME_MANAGE_LOCAL is set.

PTL_ME_EVENT_DISABLE	Specifies that this match list entry should not generate events.
PTL_ME_EVENT_SUCCESS_DISABLE	Specifies that this match list entry should not generate events that indicate success. This is useful in scenarios where the application does not need normal events, but does require failure information to enhance reliability.
PTL_ME_EVENT_OVER_DISABLE	Specifies that this match list entry should not generate overflow list events (PTL_EVENT_PUT_OVERFLOW events).
PTL_ME_EVENT_UNLINK_DISABLE	Specifies that this match list entry should not generate unlink (PTL_EVENT_UNLINK) or free (PTL_EVENT_FREE) events.
PTL_ME_EVENT_CT_GET	Enable the counting of PTL_EVENT_GET events.
PTL_ME_EVENT_CT_PUT	Enable the counting of PTL_EVENT_PUT events.
PTL_ME_EVENT_CT_PUT_OVERFLOW	Enable the counting of PTL_EVENT_PUT_OVERFLOW events.
PTL_ME_EVENT_CT_ATOMIC	Enable the counting of PTL_EVENT_ATOMIC events.
PTL_ME_EVENT_CT_ATOMIC_OVERFLOW	Enable the counting of PTL_EVENT_ATOMIC_OVERFLOW events.
PTL_ME_AUTH_USE_JID	Use job ID for authentication instead of user ID. By default, the user ID must match to allow a message to access a match list entry.
match_id	Specifies the match criteria for the process identifier of the requester. The constants PTL_PID_ANY and PTL_NID_ANY can be used to wildcard either of the physical identifiers in the **ptl_process_id_t** structure, or PTL_RANK_ANY can be used to wildcard the rank for logical addressing.
match_bits, *ignore_bits*	Specify the match criteria to apply to the match bits in the incoming request. The *ignore_bits* are used to mask out insignificant bits in the incoming match bits. The resulting bits are then compared to the match list entry's match bits to determine if the incoming request meets the match criteria.

Discussion: *Incoming match bits are compared to the match bits stored in the match list entry using the ignore bits as a mask. An optimized version of this is shown in the following code fragment:*

```
(( incoming_bits ^ match_bits ) & ~ignore_bits ) == 0
```

3.12.2 PtlMEAppend

The **PtlMEAppend()** function creates a single match list entry. If PTL_PRIORITY_LIST or PTL_OVERFLOW is specified by *ptl_list*, this entry is appended to the end of the appropriate list specified by *ptl_list* associated with the portal table entry specified by *pt_index* for the portal table for *ni_handle*. If the list is currently uninitialized, the **PtlMEAppend()** function creates the first entry in the list.

When a match list entry is posted to the priority list, the overflow list is searched to see if a matching message has arrived prior to posting the match list entry. If so, a PTL_EVENT_PUT_OVERFLOW event is generated. No searching is performed when a match list entry is posted to the overflow list.

If *ptl_list* is set to PTL_PROBE_ONLY, the overflow list is probed to support the MPI_Probe functionality. A probe of the overflow list will *always* generate a PTL_EVENT_PROBE event. If a matching message was found in the overflow list, PTL_NI_OK is returned in the event. Otherwise, the event indicates that the probe operation failed.

```
typedef enum {
    PTL_PRIORITY_LIST, PTL_OVERFLOW, PTL_PROBE_ONLY
} ptl_list_t ;
```

ME List Types

PTL_PRIORITY_LIST	The priority list associated with a portal table entry
PTL_OVERFLOW	The overflow list associated with a portal table entry
PTL_PROBE_ONLY	Do not attach to a list. Use the ME to probe the overflow list, without consuming an item in the list and without being attached anywhere.

Function Prototype for PtlMEAppend

```
int PtlMEAppend(ptl_handle_ni_t     ni_handle ,
                ptl_pt_index_t      pt_index ,
                ptl_me_t            me,
                ptl_list_t          ptl_list  ,
                void                *user_ptr ,
                ptl_handle_me_t     *me_handle);
```

Arguments

ni_handle	**input**	The interface handle to use.
pt_index	**input**	The portal table index where the match list entry should be appended.
me	**input**	Provides initial values for the user-visible parts of a match list entry. Other than its use for initialization, there is no linkage between this structure and the match list entry maintained by the API.
ptl_list	**input**	Determines whether the match list entry is appended to the priority list, appended to the overflow list, or simply queries the overflow list.
user_ptr	**input**	A user-specified value that is associated with each command that can generate an event. The value does not need to be a pointer, but must fit in the space used by a pointer. This value (along with other values) is recorded in events associated with operations on this match list entry[2].
me_handle	**output**	On successful return, this location will hold the newly created match list entry handle.

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.

[2]Tying commands to a user-defined value is useful at the target when the command needs to be associated with a data structure maintained by the process outside of the portals library. For example, an MPI implementation can set the *user_ptr* argument to the value of an MPI Request. This direct association allows for processing of match list entries by the MPI implementation without a table lookup or a search for the appropriate MPI Request.

PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_PT_INDEX_INVALID	Indicates that *pt_index* is not a valid portal table index.
PTL_PROCESS_INVALID	Indicates that *match_id* in the match list entry is not a valid process identifier.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the match list entry.
PTL_ME_LIST_TOO_LONG	Indicates that the resulting list is too long. The maximum length for a list is defined by the interface.

IMPLEMENTATION NOTE 18:

Checking *match_id*

Checking whether a *match_id* is a valid process identifier may require global knowledge. However, **PtlMEAppend()** is not meant to cause any communication with other nodes in the system. Therefore, **PTL_PROCESS_INVALID** may not be returned in some cases where it would seem appropriate.

3.12.3 PtlMEUnlink

The **PtlMEUnlink()** function can be used to unlink a match list entry from a list. This operation also releases any resources associated with the match list entry. It is an error to use the match list entry handle after calling **PtlMEUnlink()**.

Function Prototype for PtlMEUnlink

```
int PtlMEUnlink(ptl_handle_me_t me_handle);
```

Arguments

me_handle	**input**	The match list entry handle to be unlinked.

Discussion: *If this match list entry has pending operations; e.g., an unfinished reply operation, then **PtlMEUnlink()** will return **PTL_ME_IN_USE**, and the match list entry will not be unlinked. This essentially creates a race between the application retrying the unlink operation and a new operation arriving. This is believed to be reasonable as the application rarely wants to unlink an ME while new operations are arriving to it.*

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_ME_INVALID	Indicates that *me_handle* is not a valid match list entry handle.
PTL_ME_IN_USE	Indicates that the match list entry has pending operations and cannot be unlinked.

3.13 Events and Event Queues

Event queues are used to log operations performed on local match list entries or memory descriptors. In particular, they signal the end of a data transmission into or out of a memory region. They can also be used to hold acknowledgments for completed *put* operations and indicate when a match list entry has been unlinked. Multiple memory descriptors or match list entries can share a single event queue.

In addition to the **ptl_handle_eq_t** type, the portals API defines four types associated with events: The **ptl_event_kind_t** type defines the kinds of events that can be stored in an event queue. The **ptl_event_t** type defines the structure that is placed into event queues, while **ptl_initiator_event_t** and **ptl_target_event_t** types define sub-fields that hold the information associated with an event.

The portals API provides five functions for dealing with event queues: The **PtlEQAlloc()** function is used to allocate the API resources needed for an event queue, the **PtlEQFree()** function is used to release these resources, the **PtlEQGet()** function can be used to get the next event from an event queue, the **PtlEQWait()** function can be used to block a process (or thread) until an event queue has at least one event, and the **PtlEQPoll()** function can be used to test or wait on multiple event queues.

3.13.1 Kinds of Events

The portals API defines twelve types of events that can be logged in an event queue:

```
typedef enum {
    PTL_EVENT_GET,
    PTL_EVENT_PUT,
    PTL_EVENT_PUT_OVERFLOW,
    PTL_EVENT_ATOMIC,
    PTL_EVENT_REPLY,
    PTL_EVENT_SEND,
    PTL_EVENT_ACK,
    PTL_EVENT_UNLINK,
    PTL_EVENT_FREE,
    PTL_EVENT_DROPPED,
    PTL_EVENT_PROBE
} ptl_event_kind_t ;
```

Event types

`PTL_EVENT_GET`	A previously initiated *get* operation completed successfully.
`PTL_EVENT_PUT`	A previously initiated *put* operation completed successfully. The underlying layers will not alter the memory (on behalf of this operation) once this event has been logged.

PTL_EVENT_PUT_OVERFLOW	A match list entry posted by **PtlMEAppend()** matched a message that has already arrived and is managed within the overflow list. All, some, or none of the message may have been captured in local memory as requested by the match list entry and described by the *rlength* and *mlength* in the event. The event will point to the start of the message in the memory region described by the match list entry from the overflow list, if any of the message was captured. When the *rlength* and *mlength* fields do not match (i.e. the message was truncated), the application is responsible for performing the remaining transfer. This typically occurs when the application has provided an overflow list entry designed to accept headers but not message bodies. The transfer is typically done by the initiator creating a match list entry using a unique set of bits and then placing the match bits in the *hdr_data* field. The target can then use the *hdr_data* field (along with other information in the event) to retrieve the message.
PTL_EVENT_ATOMIC	A previously initiated *atomic* operation completed successfully.
PTL_EVENT_REPLY	A previously initiated *reply* operation has completed successfully . This event is logged after the data (if any) from the reply has been written into the memory descriptor.
PTL_EVENT_SEND	A previously initiated *send* operation has completed. This event is logged after the entire buffer has been sent and it is safe to reuse the buffer.
PTL_EVENT_ACK	An *acknowledgment* was received. This event is logged when the acknowledgment is received
PTL_EVENT_DROPPED	A message arrived, but did not match in the priority list and the overflow list was out of space. Thus, the message had to be dropped.
PTL_EVENT_UNLINK	A match list entry was unlinked (Section 3.12.2).
PTL_EVENT_FREE	A match list entry in the overflow list that was previously unlinked is now free to be reused by the application (Section 3.12.2).
PTL_EVENT_PROBE	A previously initiated **PtlMEAppend()** call that was set to “probe only” completed. If a match message was found in the overflow list, PTL_NI_OK is returned in the *ni_fail_type* field of the event and the event queue entries are filled in as if it were a PTL_EVENT_PUT_OVERFLOW event. Otherwise, a failure is recorded in the *ni_fail_type* field, the *user_ptr* is filled in correctly, and the other fields are undefined.

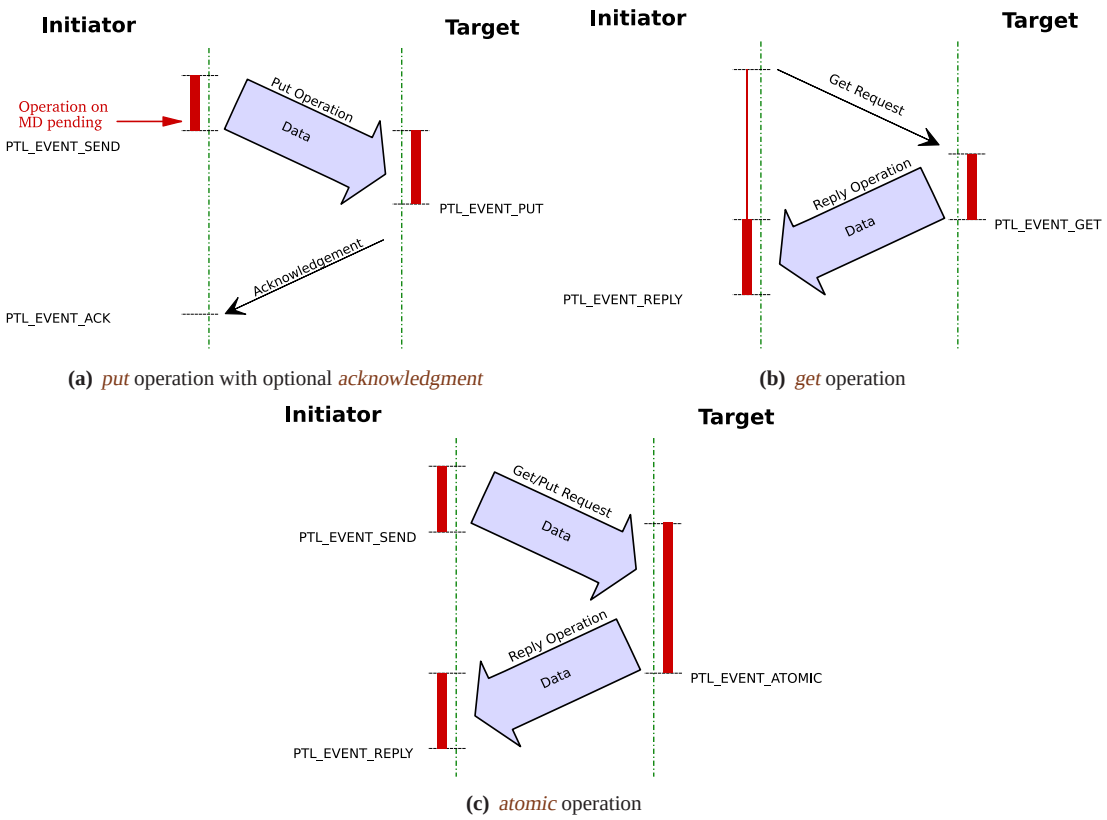
IMPLEMENTATION NOTE 19:

Overflow Events

An implementation is not required to deliver overflow events, if it can prevent an overflow from happening. For example, if an implementation used rendezvous at the lowest level, it could always choose to deliver the message into the memory of the ME that would eventually be posted

3.13.2 Event Occurrence

The diagrams in Figure 3.1 show when events occur in relation to portals operations and whether they are recorded on the *initiator* or the *target* side. Note that local and remote events are not synchronized or ordered with respect to each other.



(a) *put* operation with optional *acknowledgment*

(b) *get* operation

(c) *atomic* operation

Figure 3.1. Portals Operations and Event Types: The red bars indicate the times a local memory descriptor is considered to be in use by the system; i.e., it has operations pending. Users should not modify memory descriptors or match list entries during those periods. (Also see implementation notes 20 and 21.)

IMPLEMENTATION NOTE 20:

Pending operations and buffer modifications

Figure 3.1(a) indicates that the memory descriptor is in use from the operation iniation until PTL_EVENT_ACK. However, the initiator is free to modify the buffer the memory descriptor describes after the PTL_EVENT_SEND event. Also see implementation note 21.

Figure 3.1(a) shows the events that are generated for a *put* operation including the optional *acknowledgment*. The diagram shows which events are generated at the *initiator* and the *target* side of the *put* operation. Figure 3.1(b) shows the corresponding events for a *get* operation, and Figure 3.1(c) shows the events generated for an *atomic* operation.

If during any of the operations shown in the diagrams of Figure 3.1, a match list entry is unlinked, then a PTL_EVENT_UNLINK event is generated on the *target* where it was unlinked. This is not shown in the diagrams. None of these events are generated if the memory descriptor or match list entry has no event queue attached to it (see the description of PTL_EQ_NONE on page 45 of Section 3.10.1). The various types of events can be disabled individually. (See the description of PTL_ME_EVENT_DISABLE and PTL_ME_EVENT_UNLINK_DISABLE on page 62, also in Section 3.12.1.)

IMPLEMENTATION NOTE 21:

Pending operations and *acknowledgment*

If a user attempts to unlink a match list entry or release a memory descriptor while it has operations pending, the implementation should return **PTL_ME_IN_USE** (or **PTL_MD_IN_USE**) until the operation has completed or can be aborted cleanly.

After a PTL_EVENT_SEND a user can attempt to release the memory descriptor. If the release is successful the implementation should ensure a later acknowledgment is discarded, if it arrives. The same is true for a *reply*. Since users cannot know when events occur, the implementor has a certain amount of freedom honoring unlink requests or returning **PTL_MD_IN_USE**.

Table 3.2 summarizes the portals event types. In the table we use the word *local* to describe the location where the event is delivered; it can be the *initiator* or the *target* of an operation.

Table 3.2. Event Type Summary: A list of event types, where (*initiator* or *target*) they can occur and the meaning of those events.

Event Type	*initiator*	*target*	Meaning
PTL_EVENT_GET		•	Data was "pulled" from a local match list entry.
PTL_EVENT_PUT		•	A put matched a previously posed match list entry.
PTL_EVENT_PUT_OVERFLOW		•	A previous put arrived and matched a new match list entry.
PTL_EVENT_ATOMIC		•	Data was manipulated atomically in a local match list entry.
PTL_EVENT_ATOMIC_OVERFLOW		•	A previous atomic operation arrived and matched a new match list entry.
PTL_EVENT_REPLY	•		Data arrived at a local memory descriptor because of a local *get* or *atomic* operation.
PTL_EVENT_SEND	•		Data left a local memory descriptor because of a local *put* or *atomic* operation.
PTL_EVENT_ACK	•		An acknowledgment has arrived.
PTL_EVENT_DROPPED		•	A message was dropped because the overflow list was out of space.
PTL_EVENT_PT_DISABLED		•	A portal table entry has been disabled due to resource exhaustion.
PTL_EVENT_UNLINK		•	A local match list entry has been unlinked.
PTL_EVENT_FREE		•	A local match list entry that was posted to the overflow list and was previously is now free for reuse by the application (applies to overflow lists).
PTL_EVENT_PROBE		•	A **PtlMEAppend()** that was set to probe only completed

3.13.3 Failure Notification

There are three ways in which operations may fail to complete successfully: the system (hardware or software) can fail in a way that makes the message undeliverable, a permissions violation can occur at the target, or resources can be exhausted at a target that has enabled flow-control. In any other scenario, every operation that is started will eventually complete. While an operation is in progress, the memory on the *target* associated with the operation should not be viewed (in the case of a *put* or a *reply*) or altered on the *initiator* side (in the case of a *put* or *get*). Operation completion, whether successful or unsuccessful, is final. That is, when an operation completes, the memory associated with the operation will no longer be read or altered by the operation. A network interface can use the integral type **ptl_ni_fail_t** to define specific information regarding the failure of the operation and record this information in the *ni_fail_type* field of an event. The constant `PTL_NI_OK` should be used in successful end events to indicate that there has been no failure. In turn, the constant `PTL_NI_UNDELIVERABLE` should indicate a system failure that prevents message delivery. The constant `PTL_NI_FLOW_CTRL` should indicate that the remote node has exhausted its resources and has enabled flow control and dropped this message. The constant `PTL_NI_PERM_VIOLATION` should indicate that the remote Portals addressing has indicated a permissions violation for this message. The latter two error types require the stateful delivery of information from the target, and can be disabled by using `PTL_MD_REMOTE_FAILURE_DISABLE` in the MD options (see Section 3.10).

IMPLEMENTATION NOTE 22:

Completion of portals operations

Portals guarantees that every operation started will finish with an event if events are not disabled. While this document cannot enforce or recommend a suitable time, a quality implementation will keep the amount of time between an operation initiation and a corresponding event as short as possible. That includes operations that do not complete successfully. Timeouts of underlying protocols should be chosen accordingly

3.13.4 The Event Queue Types

An event queue contains **ptl_event_t** structures, which contain a *type* and a union of the *target* specific event structure and the *initiator* specific event structure.

```
typedef struct {
    ptl_event_kind_t        type;
    union {
        ptl_target_event_t      tevent;
        ptl_initiator_event_t   ievent;
    } event;
} ptl_event_t;
```

Members

type	Indicates the type of the event.
event	Contains the event information.

An operation on the *target* needs information about the local match list entry modified, the initiator of the operation and the operation itself. These fields are included in a strcuture:

```
typedef struct {
    ptl_process_id_t        initiator ; /* nid, pid or rank */
    ptl_pt_index_t          pt_index ;
    ptl_uid_t               uid;
    ptl_jid_t               jid ;
    ptl_match_bits_t        match_bits ;
    ptl_size_t              rlength ;
    ptl_size_t              mlength;
    ptl_size_t              remote_offset ;
    void                    *start ;
    void                    *user_ptr ;
    ptl_hdr_data_t          hdr_data;
    ptl_ni_fail_t           ni_fail_type ;
    ptl_op_t                atomic_operation ;
    ptl_datatype_t          atomic_type ;
    volatile ptl_seq_t      sequence;
} ptl_target_event_t ;
```

Members

initiator	The identifier of the *initiator* (**ptl_process_id_t**).
pt_index	The portal table index where the message arrived.
uid	The user identifier of the *initiator*.
jid	The job identifier of the *initiator*. May be `PTL_JID_NONE` in implementations that do not support job identifiers.
match_bits	The match bits specified by the *initiator*.
rlength	The length (in bytes) specified in the request.
mlength	The length (in bytes) of the data that was manipulated by the operation. For truncated operations, the manipulated length will be the number of bytes specified by the memory descriptor operation (possibly with an offset). For all other operations, the manipulated length will be the length of the requested operation.
remote_offset	The offset requested by the initiator.
start	The starting location (virtual, byte address) where the message has been placed. The *start* variable is the sum of the *start* variable in the match list entry and the offset used for the operation. The offset can be determined by the operation (Section 3.15) for a remote managed match list entry or by the local memory descriptor (Section 3.12). When the **PtlMEAppend()** call matches a message that has arrived in the overflow list, the start address points to the address in the overflow list where the matching message resides. This may require the application to copy the message to the desired buffer.
user_ptr	A user-specified value that is associated with each command that can generate an event. The *user_ptr* is placed in the event. For further discussion of *user_ptr*, see Section 3.12.2.

hdr_data	64 bits of out-of-band user data (Section 3.15.2).
ni_fail_type	Is used to convey the failure of an operation. Success is indicated by `PTL_NI_OK`; see section 3.13.3.
atomic_operation	If this event corresponds to an atomic operation, this indicates the atomic operation that was performed
atomic_type	If this event corresponds to an atomic operation, this indicates the data type of the atomic operation that was performed
sequence	The sequence number for this event. Sequence numbers are unique to each event.

The *initiator*, in contrast, can track all information about the attempted operation; however, it does need the result of the operation and a pointer to resolve back to the local structure tracking the information about the operation. These fields are provided by a much smaller event structure:

```
typedef struct {
    ptl_size_t          mlength;
    ptl_size_t          offset ;
    void                *user_ptr ;
    ptl_ni_fail_t       ni_fail_type ;
    volatile ptl_seq_t  sequence;
} ptl_initiator_event_t ;
```

Members

mlength, *ni_fail_type*, *sequence*, *user_ptr*	See the discussion of **ptl_target_event_t**.
offset	The displacement (in bytes) into the memory region that the operation used. The offset can be determined by the operation (Section 3.15) for a remote managed memory descriptor or by the local memory descriptor (Section 3.10). The offset and the length of the memory descriptor can be used to determine if *min_free* has been exceeded.

Discussion: *The sequence member is the last member and is volatile to support shared memory processor (SMP) implementations. When a portals implementation fills in an event structure, the sequence member should be written after all other members have been updated. Moreover, a memory barrier should be inserted between the updating of other members and the updating of the sequence member.*

3.13.5 PtlEQAlloc

The **PtlEQAlloc()** function is used to build an event queue.

Function Prototype for PtlEQAlloc

```
int PtlEQAlloc(ptl_handle_ni_t      ni_handle ,
               ptl_size_t           count,
               ptl_handle_eq_t      *eq_handle);
```

Arguments

ni_handle	**input**	The interface handle with which the event queue will be associated.
count	**input**	A hint as to the number of events to be stored in the event queue. An implementation may provide space for more than the requested number of event queue slots.
eq_handle	**output**	On successful return, this location will hold the newly created event queue handle.

Discussion: *An event queue has room for at least count number of events. The event queue is circular. If flow control is not enabled on the portal table entry (Sections 3.6.1 and 2.3, then older events will be overwritten by new ones if they are not removed in time by the user — using the functions* ***PtlEQGet()****,* ***PtlEQWait()****, or* ***PtlEQPoll()****. It is up to the user to determine the appropriate size of the event queue to prevent this loss of events.*

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the event queue.
PTL_SEGV	Indicates that *eq_handle* is not a legal address.

IMPLEMENTATION NOTE 23: Location of event queue

The event queue is designed to reside in user space. High-performance implementations can be designed so they only need to write to the event queue but never have to read from it. This limits the number of protection boundary crossings to update the event queue. However, implementors are free to place the event queue anywhere they like; inside the kernel or the NIC for example.

IMPLEMENTATION NOTE 24: Size of event queue and reserved space

Because flow control may be enabled on the portal table entries that this EQ is attached to, the implementation should insure that the space allocated for the EQ is large enough to hold the requested number of events plus the number of portal table entries associated with this *ni_handle*. For each **PtlPTAlloc()** that enables flow control and uses a given EQ, one space should be reserved for a `PTL_EVENT_PT_DISABLED` event associated with that EQ.

3.13.6 PtlEQFree

The **PtlEQFree()** function releases the resources associated with an event queue. It is up to the user to ensure that no memory descriptors or match list entries are associated with the event queue once it is freed.

Function Prototype for PtlEQFree

```
int  PtlEQFree(ptl_handle_eq_t      eq_handle);
```

Arguments

eq_handle	**input**	The event queue handle to be released.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_EQ_INVALID	Indicates that *eq_handle* is not a valid event queue handle.

3.13.7 PtlEQGet

The **PtlEQGet()** function is a nonblocking function that can be used to get the next event in an event queue. The event is removed from the queue.

Function Prototype for PtlEQGet

```
int  PtlEQGet(ptl_handle_eq_t      eq_handle,
              ptl_event_t          *event);
```

Arguments

eq_handle	**input**	The event queue handle.
event	**output**	On successful return, this location will hold the values associated with the next event in the event queue.

Return Codes

PTL_OK	Indicates success.
PTL_EQ_DROPPED	Indicates success (i.e., an event is returned) and that at least one event between this event and the last event obtained — using **PtlEQGet()**, **PtlEQWait()**, or **PtlEQPoll()**— from this event queue has been dropped due to limited space in the event queue.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_EQ_EMPTY	Indicates that *eq_handle* is empty or another thread is waiting in **PtlEQWait()**.
PTL_EQ_INVALID	Indicates that *eq_handle* is not a valid event queue handle.
PTL_SEGV	Indicates that *event* is not a legal address.

3.13.8 PtlEQWait

The **PtlEQWait()** function can be used to block the calling process or thread until there is an event in an event queue. This function returns the next event in the event queue and removes this event from the queue. In the event that multiple threads are waiting on the same event queue, **PtlEQWait()** is guaranteed to wake exactly one thread, but the order in which they are awakened is not specified.

Function Prototype for PtlEQWait

```
int  PtlEQWait(ptl_handle_eq_t      eq_handle,
               ptl_event_t          *event);
```

Arguments

eq_handle	**input**	The event queue handle to wait on. The calling process (thread) will be blocked until the event queue is not empty.
event	**output**	On successful return, this location will hold the values associated with the next event in the event queue.

Return Codes

PTL_OK	Indicates success.
PTL_EQ_DROPPED	Indicates success (i.e., an event is returned) and that at least one event between this event and the last event obtained — using **PtlEQGet()**, **PtlEQWait()**, or **PtlEQPoll()**— from this event queue has been dropped due to limited space in the event queue.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_EQ_INVALID	Indicates that *eq_handle* is not a valid event queue handle.
PTL_SEGV	Indicates that *event* is not a legal address.

3.13.9 PtlEQPoll

The **PtlEQPoll()** function can be used by the calling process to look for an event from a set of event queues. Should an event arrive on any of the queues contained in the array of event queue handles, the event will be returned in *event* and *which* will contain the index of the event queue from which the event was taken.

If **PtlEQPoll()** returns success, the corresponding event is consumed. **PtlEQPoll()** provides a timeout to allow applications to poll, block for a fixed period, or block indefinitely. **PtlEQPoll()** is sufficiently general to implement both **PtlEQGet()** and **PtlEQWait()**, but these functions have been retained in the API for backward compatibility.

IMPLEMENTATION NOTE 25:

Fairness of **PtlEQPoll()**

PtlEQPoll() should poll the list of queues in a round-robin fashion. This cannot guarantee fairness but meets common expectations.

Function Prototype for PtlEQPoll

```
int  PtlEQPoll(ptl_handle_eq_t      *eq_handles,
               int                   size ,
               ptl_time_t            timeout,
               ptl_event_t     *event,
               int                   *which);
```

Arguments

eq_handles	**input**	An array of event queue handles. All the handles must refer to the same interface.
size	**input**	Length of the array.
timeout	**input**	Time in milliseconds to wait for an event to occur on one of the event queue handles. The constant `PTL_TIME_FOREVER` can be used to indicate an infinite timeout.
event	**output**	On successful return (**PTL_OK** or **PTL_EQ_DROPPED**), this location will hold the values associated with the next event in the event queue.
which	**output**	On successful return, this location will contain the index into *eq_handles* of the event queue from which the event was taken.

Return Codes

PTL_OK	Indicates success.
PTL_EQ_DROPPED	Indicates success (i.e., an event is returned) and that at least one event between this event and the last event obtained from the event queue indicated by *which* has been dropped due to limited space in the event queue.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_EQ_INVALID	Indicates that one or more of the event queue handles is not valid; e.g., not all handles in *eq_handles* are on the same network interface.
PTL_SEGV	Indicates that *event* or *which* is not a legal address.
PTL_EQ_EMPTY	Indicates that the timeout has been reached and all of the event queues are empty.

IMPLEMENTATION NOTE 26: Macros using **PtlEQPoll()**

Implementations are free to provide macros for **PtlEQGet()** and **PtlEQWait()** that use **PtlEQPoll()** instead of providing these functions.

IMPLEMENTATION NOTE 27: Filling in the **ptl_event_t** and **ptl_target_event_t** structures

All of the members of the **ptl_event_t** structure (and corresponding **ptl_initiator_event_t** or **ptl_target_event_t** sub-field) returned from **PtlEQGet()**, **PtlEQWait()**, and **PtlEQPoll()** must be filled in with valid information. An implementation may not leave any field in an event unset.

3.14 Lightweight “Counting” Events

Standard events copy a significant amount of data from the implementation to the application. While this data is critical for many uses (e.g. MPI), other programming models (e.g. PGAS) require very little information about individual operations. To support lightweight operations, Portals provide a lightweight event mechanism known as counting events.

Counting events are enabled by attaching an **ptl_handle_ct_t** to a memory descriptor or match list entry and by specifying which operations are to be counted in the options field. Counting events can be set to count either the total number of operations *or* the number of bytes transfered for the associated operations.

Counting events mirror standard events in virtually every way. They can be used to log the same set of operations performed on local match list entries or memory descriptors that event queues log. Counting events introduce an additional type — the counting event handle: **ptl_handle_ct_t**. A **ptl_handle_ct_t** refers two unsigned 64-bit integral type variables that are allocated through a **PtlCTAlloc()**, queried through a **PtlCTGet()** or **PtlCTWait()**, set through a **PtlCTSet()**, incremented through a **PtlCTInc()**, and freed through a **PtlCTFree()**. To mirror the failure semantics of the standard events, one variable counts the successful events and the second variable counts the events that failed.

IMPLEMENTATION NOTE 28: Counting Event Handles

A high performance implementation could choose to make a **ptl_handle_ct_t** a simple pointer to a structure in the address space of the application; however, in some cases, it may be desirable, or even necessary, to allocate these pointers in a special part of the address space (e.g. low physical addresses to facilitate accesses by particular hardware).

Semantics for event occurrence match those described in Sections 3.13.2. They can be independently enabled/disabled with options on the memory descriptor or match list entry analogous to those used for event queues.

3.14.1 The Counting Event Type

A *ct_handle* refers to a **ptl_ct_event_t** structure. The user visible portion of this structure contains both a count of succeeding events and a count of failing events.

```
typedef struct {
    ptl_size_t      success;
    ptl_size_t      failure ;
} ptl_ct_event_t ;
```

Members

success	A count associated with successful events that counts events or bytes.
failure	A count associated with failed events that counts events or bytes.

3.14.2 PtlCTAlloc

The **PtlCTAlloc()** function is used to allocate a counting event that counts either operations on the memory descriptor (match list entry) or bytes that flow out of (into) a memory descriptor (match list entry). While a **PtlCTAlloc()** call could be as simple as a malloc of a structure holding the counting event and a network interface handle, it may be necessary to allocate the counting event in low memory or some other protected space; thus, an allocation routine is provided. A newly allocated count is initialized to zero.

```
typedef enum {
    PTL_CT_OPERATION, PTL_CT_BYTE
} ptl_ct_type_t ;
```

Function Prototype for PtlCTAlloc

```
int PtlCTAlloc(ptl_handle_ni_t     ni_handle ,
               ptl_ct_type_t       ct_type ,
               ptl_handle_ct_t     *ct_handle );
```

Arguments

ni_handle	**input**	The interface handle with which the counting event will be associated.
ct_type	**input**	A selection between counting operations and counting bytes.
ct_handle	**output**	On successful return, this location will hold the newly created counting event handle.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the counting event.
PTL_SEGV	Indicates that *ct_handle* is not a legal address.

IMPLEMENTATION NOTE 29:

Minimizing cost of counting events

A quality implementation will attempt to minimize the cost of counting events. This can be done by translating the simple functions (**PtlCTGet()**, **PtlCTWait()**, **PtlCTSet()**, and **PtlCTInc()**) into simple macros that directly access a structure in the applications memory unless otherwise required by the hardware.

3.14.3 PtlCTFree

The **PtlCTFree()** function releases the resources associated with a counting event. It is up to the user to ensure that no memory descriptors or match list entries are associated with the counting event once it is freed.

Function Prototype for PtlCTFree

```
int PtlCTFree(ptl_handle_ct_t ct_handle );
```

Arguments

ct_handle	**input**	The counting event handle to be released.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_CT_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.14.4 PtlCTGet

The **PtlCTGet()** function is used to obtain the current value of a counting event.

Function Prototype for PtlCTGet

```
int PtlCTGet(ptl_handle_ct_t ct_handle ,
             ptl_ct_event_t *event );
```

Arguments

ct_handle	**input**	The counting event handle.
event	**output**	On successful return, this location will hold the current value associated with the counting event.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_CT_INVALID	Indicates that *ct_handle* is not a valid counting event handle.
PTL_SEGV	Indicates that *event* is not a legal address.

3.14.5 PtlCTWait

The **PtlCTWait()** function is used to wait until the value of a counting event is equal to a test value.

Function Prototype for PtlCTWait

```
int  PtlCTWait(ptl_handle_ct_t      ct_handle ,
               ptl_size_t           test );
```

Arguments

ct_handle	**input**	The counting event handle.
test	**input**	On successful return, the sum of the success and failure fields of the counting event will be greater than or equal to this value.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_CT_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.14.6 PtlCTSet

Periodically, it is desirable to reinitialize or adjust the value of a counting event. The **PtlCTSet()** function is used to set the value of a counting event.

Function Prototype for PtlCTSet

```
int  PtlCTSet(ptl_handle_ct_t      ct_handle ,
              ptl_ct_event_t       new_ct);
```

Arguments

ct_handle	**input**	The counting event handle.
new_ct	**input**	On successful return, the value of the counting event will have been set to this value.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_CT_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.14.7 PtlCTInc

In some scenarios, the counting event will need to be incremented by the application. This must be done atomically, so a functional interface is provided. The **PtlCTInc()** function is used to increment the value of a counting event.

> **Discussion**: *As an example, a counting event may need to be incremented at the completion of a message that is received. If the message arrives in the overflow list, it may be desirable to delay the counting event increment until the application can place the data in the correct buffer.*

Function Prototype for PtlCTInc

```
int  PtlCTInc(ptl_handle_ct_t      ct_handle ,
              ptl_ct_event_t       increment);
```

Arguments

ct_handle	**input**	The counting event handle.
increment	**input**	On successful return, the value of the counting event will have been incremented by this value.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_CT_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.15 Data Movement Operations

The portals API provides five data movement operations: **PtlPut()**, **PtlGet()**, **PtlAtomic()**, **PtlFetchAtomic()**, and **PtlSwap()**.

> **IMPLEMENTATION NOTE 30:** Functions that require communication
>
> Other than **PtlPut()**, **PtlGet()**, **PtlAtomic()**, **PtlFetchAtomic()**, and **PtlSwap()** (and their triggered variants), no function in the portals API requires communication with other nodes in the system.

3.15.1 Portals Acknowledgment Type Definition

Values of the type **ptl_ack_req_t** are used to control whether an acknowledgment should be sent when the operation completes (i.e., when the data has been written to a match list entry of the *target* process). The value `PTL_ACK_REQ` requests an acknowledgment, the value `PTL_NO_ACK_REQ` requests that no acknowledgment should be generated, the value `PTL_CT_ACK_REQ` requests a simple counting acknowledgment, and the value `PTL_OC_ACK_REQ` requests an

operation completed acknowledgement. When a counting acknowledgment is requested, either PTL_CT_OPERATION or PTL_CT_BYTE can be set in the *ct_handle*. If PTL_CT_OPERATION is set, the number of acknowledgments is counted. If PTL_CT_BYTE is set, the modified length (*mlength*) from the target is counted at the initiator. The operation completed acknowledgement is an acknowledgement that simply indicated that the operation has completed at the target. It *does not* indicate what was done with the message. The message may have been dropped due to a permission violation or may not have matched in the priority list or overflow list; however, the operation completed acknowledgement would still be sent. The operation completed acknowledgement is a subset of the counting acknowledgement with weaker semantics. That is, it is a counting type of acknowledgement, but it can only count operations.

```
typedef enum {PTL_ACK_REQ,
              PTL_NO_ACK_REQ,
              PTL_CT_ACK_REQ,
              PTL_OC_ACK_REQ
} ptl_ack_req_t ;
```

3.15.2 PtlPut

The **PtlPut()** function initiates an asynchronous *put* operation. There are several events associated with a *put* operation: completion of the send on the *initiator* node (PTL_EVENT_SEND) and, when the send completes successfully, the receipt of an acknowledgment (PTL_EVENT_ACK) indicating that the operation was accepted by the *target*. The event PTL_EVENT_PUT is used at the *target* node to indicate the end of data delivery, while PTL_EVENT_PUT_OVERFLOW can be used on the *target* node when a message arrives before the corresponding match list entry (Figure 3.1).

These (local) events will be logged in the event queue associated with the memory descriptor (*md_handle*) used in the *put* operation. Using a memory descriptor that does not have an associated event queue results in these events being discarded. In this case, the caller must have another mechanism (e.g., a higher level protocol) for determining when it is safe to modify the memory region associated with the memory descriptor.

The local (*initiator*) offset is used to determine the starting address of the memory region within the region specified by the memory descriptor and the length specifies the length of the region in bytes. It is an error for the local offset and length parameters to specify memory outside the memory described by the memory descriptor.

Function Prototype for PtlPut

```
int  PtlPut (ptl_handle_md_t  md_handle,
             ptl_size_t       local_offset ,
             ptl_size_t       length ,
             ptl_ack_req_t    ack_req ,
             ptl_process_id_t target_id ,
             ptl_pt_index_t   pt_index ,
             ptl_match_bits_t match_bits ,
             ptl_size_t       remote_offset ,
             void             *user_ptr ,
             ptl_hdr_data_t   hdr_data);
```

Arguments

md_handle	**input**	The memory descriptor handle that describes the memory to be sent. If the memory descriptor has an event queue associated with it, it will be used to record events when the message has been sent (`PTL_EVENT_SEND`, `PTL_EVENT_ACK`).
local_offset	**input**	Offset from the start of the memory descriptor.
length	**input**	Length of the memory region to be sent.
ack_req	**input**	Controls whether an acknowledgment event is requested. Acknowledgments are only sent when they are requested by the initiating process **and** the memory descriptor has an event queue **and** the target memory descriptor enables them. Allowed constants: `PTL_ACK_REQ`, `PTL_NO_ACK_REQ`, `PTL_CT_ACK_REQ`, `PTL_OC_ACK_REQ`.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
match_bits	**input**	The match bits to use for message selection at the *target* process (only used when matching is enabled on the network interface).
remote_offset	**input**	The offset into the target memory descriptor (used unless the *target* match list entry has the `PTL_ME_MANAGE_LOCAL` option set).
user_ptr	**input**	A user-specified value that is associated with each command that can generate an event. The value does not need to be a pointer, but must fit in the space used by a pointer. This value (along with other values) is recorded in *initiator* events associated with this *put* operation[3].
hdr_data	**input**	64 bits of user data that can be included in the message header. This data is written to an event queue entry at the *target* if an event queue is present on the match list entry that matches the message.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.

3.15.3 PtlGet

The **PtlGet()** function initiates a remote read operation. There are two events associated with a get operation. When the data is sent from the *target* node, a `PTL_EVENT_GET` event is registered on the *target* node. When the data is returned from the *target* node, a `PTL_EVENT_REPLY` event is registered on the *initiator* node. (Figure 3.1)

The local (*initiator*) offset is used to determine the starting address of the memory region and the length specifies the length of the region in bytes. It is an error for the local offset and length parameters to specify memory outside the memory described by the memory descriptor.

[3]Tying commands to a user-defined value is useful for quickly locating a user data structure associated with the *put* operation. For example, an MPI implementation can set the *user_ptr* argument to the value of an MPI Request. This direct association allows for processing of a *put* operation completion event by the MPI implementation without a table lookup or a search for the appropriate MPI Request.

Function Prototype for PtlGet

```
int  PtlGet(ptl_handle_md_t   md_handle,
            ptl_size_t        local_offset  ,
            ptl_size_t        length ,
            ptl_process_id_t  target_id  ,
            ptl_pt_index_t    pt_index ,
            ptl_match_bits_t  match_bits ,
            void             *user_ptr ,
            ptl_size_t        remote_offset  );
```

Arguments

md_handle	**input**	The memory descriptor handle that describes the memory into which the requested data will be received. The memory descriptor can have an event queue associated with it to record events, such as when the message receive has started.
local_offset	**input**	Offset from the start of the memory descriptor.
length	**input**	Length of the memory region for the *reply*.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
match_bits	**input**	The match bits to use for message selection at the *target* process.
user_ptr	**input**	See the discussion for **PtlPut()**.
remote_offset	**input**	The offset into the target match list entry (used unless the target match list entry has the `PTL_ME_MANAGE_LOCAL` option set).

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.

3.15.4 Portals Atomics Overview

Portals defines three closely related types of atomic operations. The **PtlAtomic()** function is a one-way operation that performs an atomic operation on data at the *target* with the data passed in the *put* memory descriptor. The **PtlFetchAtomic()** function extends **PtlAtomic()** to be an atomic fetch-and-update operation; thus, the value at the *target* before the operation is returned in a *reply* message and placed into the *get* memory descriptor of the *initiator*. Finally, the **PtlSwap()** operation atomically swaps data (including compare-and-swap and swap under mask, which require an *operand* argument).

The length of the operations performed by a **PtlAtomic()** or **PtlFetchAtomic()** is restricted to no more than *max_atomic_size* bytes. **PtlSwap()** operations can also be up to *max_atomic_size* bytes, except for `PTL_CSWAP` and `PTL_MSWAP` operations, which are further restricted to 8 bytes (the length of the longest native data type) in all

implementations. The *target* match list entry must be configured to respond to *put* operations and to *get* operations if a reply is desired. The *length* argument at the initiator is used to specify the size of the request.

There are three events that can be associated with atomic operations. When data is sent from the *initiator* node, a PTL_EVENT_SEND event is registered on the *initiator* node. If data is sent from the *target* node, a PTL_EVENT_ATOMIC event is registered on the *target* node; and if data is returned from the *target* node, a PTL_EVENT_REPLY event is registered on the *initiator* node. Note that the target match list entry must have the PTL_ME_OP_PUT flag set and must also set the PTL_ME_OP_GET flag to enable a reply.

The three atomic functions share two new arguments introduced in Portals 4.0: an operation (**ptl_op_t**) and a datatype (**ptl_datatype_t**), as described below.

```
typedef enum {
    PTL_MIN, PTL_MAX,
    PTL_SUM, PTL_PROD,
    PTL_LOR, PTL_LAND,
    PTL_BOR, PTL_BAND,
    PTL_LXOR, PTL_BXOR,
    PTL_SWAP, PTL_CSWAP, PTL_MSWAP
} ptl_op_t ;
```

Atomic Operations

PTL_MIN	Compute and return the minimum of the initiator and target value.
PTL_MAX	Compute and return the maximum of the initiator and target value.
PTL_SUM	Compute and return the sum of the initiator and target value.
PTL_PROD	Compute and return the product of the initiator and target value.
PTL_LOR	Compute and return the logical OR of the initiator and target value.
PTL_LAND	Compute and return the logical AND of the initiator and target value.
PTL_BOR	Compute and return the bitwise OR of the initiator and target value.
PTL_BAND	Compute and return the bitwise AND of the initiator and target value.
PTL_LXOR	Compute and return the logical XOR of the initiator and target value.
PTL_BXOR	Compute and return the bitwise XOR of the initiator and target value.
PTL_SWAP	Swap the initiator and target value and return the target value.
PTL_CSWAP	A conditional swap — if the value of the operand is equal to the target value, the initiator and target value are swapped. The target value is always returned. This operation is limited to single data items.
PTL_MSWAP	A swap under mask — update the bits of the target value that are set to 1 in the operand and return the target value. This operation is limited to single data items.

```
typedef enum {
    PTL_CHAR, PTL_UCHAR,
    PTL_SHORT, PTL_USHORT,
    PTL_INT, PTL_UINT,
    PTL_LONG, PTL_ULONG,
    PTL_FLOAT, PTL_DOUBLE
} ptl_datatype_t ;
```

Atomic Datatypes

PTL_CHAR	8-bit signed integer
PTL_UCHAR	8-bit unsigned integer
PTL_SHORT	16-bit signed integer
PTL_USHORT	16-bit unsigned integer
PTL_INT	32-bit signed integer
PTL_UINT	32-bit unsigned integer
PTL_LONG	64-bit signed integer
PTL_ULONG	64-bit unsigned integer
PTL_FLOAT	32-bit floating-point number
PTL_DOUBLE	64-bit floating-point number

3.15.5 PtlAtomic

Function Prototype for PtlAtomic

```
int PtlAtomic(ptl_handle_md_t    md_handle,
              ptl_size_t         local_offset ,
              ptl_size_t         length ,
              ptl_ack_req_t      ack_req ,
              ptl_process_id_t   target_id ,
              ptl_pt_index_t     pt_index ,
              ptl_match_bits_t   match_bits ,
              ptl_size_t         remote_offset ,
              void               *user_ptr ,
              ptl_hdr_data_t     hdr_data ,
              ptl_op_t           operation ,
              ptl_datatype_t     datatype );
```

Arguments

md_handle	**input**	The memory descriptor handle that describes the memory to be sent. If the memory descriptor has an event queue associated with it, it will be used to record events when the message has been sent.
local_offset	**input**	Offset from the start of the memory descriptor referenced by the *md_handle* to use for transmitted data.

length	**input**	Length of the memory region to be sent and/or received.
ack_req	**input**	Controls whether an acknowledgment event is requested. Acknowledgments are only sent when they are requested by the initiating process **and** the memory descriptor has an event queue **and** the target memory descriptor enables them. Allowed constants: `PTL_ACK_REQ`, `PTL_NO_ACK_REQ`, `PTL_CT_ACK_REQ`, `PTL_OC_ACK_REQ`.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
match_bits	**input**	The match bits to use for message selection at the *target* process.
remote_offset	**input**	The offset into the target memory descriptor (used unless the target memory descriptor has the `PTL_ME_MANAGE_LOCAL` option set).
user_ptr	**input**	See the discussion for **PtlPut()**.
hdr_data	**input**	64 bits of user data that can be included in the message header. This data is written to an event queue entry at the *target* if an event queue is present on the match list entry that the message matches.
operation	**input**	The operation to be performed using the initiatior and target data.
datatype	**input**	The type of data being operated on at the initiatior and target.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.

3.15.6 PtlFetchAtomic

Function Prototype for PtlFetchAtomic

```
int PtlFetchAtomic(ptl_handle_md_t      get_md_handle,
                   ptl_size_t           local_get_offset ,
                   ptl_handle_md_t      put_md_handle,
                   ptl_size_t           local_put_offset ,
                   ptl_size_t           length ,
                   ptl_process_id_t     target_id ,
                   ptl_pt_index_t       pt_index ,
                   ptl_match_bits_t     match_bits ,
                   ptl_size_t           remote_offset ,
                   void                 *user_ptr ,
                   ptl_hdr_data_t       hdr_data,
                   ptl_op_t             operation ,
                   ptl_datatype_t       datatype );
```

Arguments

get_md_handle	**input**	The memory descriptor handle that describes the memory into which the result of the operation will be placed. The memory descriptor can have an event queue associated with it to record events, such as when the result of the operation has been returned.
local_get_offset	**input**	Offset from the start of the memory descriptor referenced by the *get_md_handle* to use for received data.
put_md_handle	**input**	The memory descriptor handle that describes the memory to be sent. If the memory descriptor has an event queue associated with it, it will be used to record events when the message has been sent.
local_put_offset	**input**	Offset from the start of the memory descriptor referenced by the *put_md_handle* to use for transmitted data.
length	**input**	Length of the memory region to be sent and/or received.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
match_bits	**input**	The match bits to use for message selection at the *target* process.
remote_offset	**input**	The offset into the target memory descriptor (used unless the target memory descriptor has the `PTL_ME_MANAGE_LOCAL` option set).
user_ptr	**input**	See the discussion for **PtlPut()**.
hdr_data	**input**	64 bits of user data that can be included in the message header. This data is written to an event queue entry at the *target* if an event queue is present on the match list entry that the message matches.
operation	**input**	The operation to be performed using the initiatior and target data.
datatype	**input**	The type of data being operated on at the initiatior and target.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *put_md_handle* or *get_md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.

3.15.7 PtlSwap

Function Prototype for PtlSwap

```
int  PtlSwap(ptl_handle_md_t      get_md_handle,
             ptl_size_t           local_get_offset ,
             ptl_handle_md_t      put_md_handle,
             ptl_size_t           local_put_offset ,
             ptl_size_t           length ,
             ptl_process_id_t     target_id ,
             ptl_pt_index_t       pt_index ,
             ptl_match_bits_t     match_bits ,
             ptl_size_t           remote_offset ,
             void                 *user_ptr ,
             ptl_hdr_data_t       hdr_data,
             void                 *operand,
             ptl_op_t             operation ,
             ptl_datatype_t       datatype );
```

Arguments

get_md_handle	**input**	The memory descriptor handle that describes the memory into which the result of the operation will be placed. The memory descriptor can have an event queue associated with it to record events, such as when the result of the operation has been returned.
local_get_offset	**input**	Offset from the start of the memory descriptor referenced by the *get_md_handle* to use for received data.
put_md_handle	**input**	The memory descriptor handle that describes the memory to be sent. If the memory descriptor has an event queue associated with it, it will be used to record events when the message has been sent.
local_put_offset	**input**	Offset from the start of the memory descriptor referenced by the *put_md_handle* to use for transmitted data.
length	**input**	Length of the memory region to be sent and/or received.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
match_bits	**input**	The match bits to use for message selection at the *target* process.
remote_offset	**input**	The offset into the target memory descriptor (used unless the target memory descriptor has the `PTL_ME_MANAGE_LOCAL` option set).
user_ptr	**input**	See the discussion for **PtlPut()**.
hdr_data	**input**	64 bits of user data that can be included in the message header. This data is written to an event queue entry at the *target* if an event queue is present on the match list entry that the message matches.
operand	**input**	A pointer to the data to be used for the `PTL_CSWAP` and `PTL_MSWAP` operations (ignored for other operations). The data pointed to is of the type specified by the *datatype* argument and must be included in the message.
operation	**input**	The operation to be performed using the initiatior and target data.
datatype	**input**	The type of data being operated on at the initiatior and target.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *put_md_handle* or *get_md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.

3.16 Triggered Operations

For a variety of scenarios, it is desirable to setup a response to incoming messages. As an example, a tree based reduction operation could be performed by having each layer of the tree issue a **PtlAtomic()** operation to its parent after receiving a **PtlAtomic()** from all of its children. To provide this operation, triggered versions of each of the data movement operations are provided. To create a triggered operation, a *trig_ct_handle* and an integer *threshold* are added to the argument list. When the count referenced by the *trig_ct_handle* argument reaches or exceeds the *threshold* (equal to or greater), the operation proceeds *at the initiator of the operation*. For example, a **PtlTriggeredGet()** or a **PtlTriggeredAtomic()** will not leave the *initiator* until the threshold is reached.

Discussion: *The use of a trig_ct_handle and threshold enables a variety of usage models. A single match list entry can trigger one operation (or several) by using an independent trig_ct_handle on the match list entry. One operation can be triggered by a combination of previous events (include a combination of initiator and target side events) by having all of the earlier operations reference a single trig_ct_handle and using an appropriate threshold.*

IMPLEMENTATION NOTE 31: Ordering of Triggered Operations

The semantics of triggered operations imply that (at a minimum) operations will proceed in the order that their trigger threshold is reached. A quality implementation will also release operations that reach their threshold simultaneously on the same *trig_ct_handle* in the order that they are issued.

IMPLEMENTATION NOTE 32: Implementation of Triggered Operations

The most straightforward way to implement triggered operations is to associate a list of dependent operations with the structure referenced by a *trig_ct_handle*. Operations depending on the same *trig_ct_handle* with the same *threshold* *should* proceed in the order that they were issued; thus, the list of operations associated with a *trig_ct_handle* may be sorted for faster searching.

IMPLEMENTATION NOTE 33: Triggered Operations Reaching the Threshold

The triggered operation is released when the counter referenced by the *trig_ct_handle* reaches or exceeds the *threshold*. This means that the triggered operation must check the value of the *trig_ct_handle* in an atomic way when it is first associated with the *trig_ct_handle*.

3.16.1 PtlTriggeredPut

The **PtlTriggeredPut()** function adds triggered operation semantics to the **PtlPut()** function described in Section 3.15.2.

Function Prototype for PtlTriggeredPut

```
int PtlTriggeredPut (ptl_handle_md_t md_handle,
                     ptl_size_t local_offset ,
                     ptl_size_t length ,
                     ptl_ack_req_t ack_req ,
                     ptl_process_id_t target_id ,
                     ptl_pt_index_t pt_index ,
                     ptl_match_bits_t match_bits ,
                     ptl_size_t remote_offset ,
                     void *user_ptr ,
                     ptl_hdr_data_t hdr_data,
                     ptl_handle_ct_t trig_ct_handle ,
                     ptl_size_t threshold );
```

Arguments

md_handle, *local_offset*, *length*, *ack_req*, *target_id*, *pt_index*, *match_bits*, *remote_offset*, *user_ptr*, *hdr_data*	**input**	See description in Section 3.15.2.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.
PTL_CT_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.16.2 PtlTriggeredGet

The **PtlTriggeredGet()** function adds triggered operation semantics to the **PtlGet()** function described in Section 3.15.3.

Function Prototype for PtlTriggeredGet

```
int  PtlTriggeredGet (ptl_handle_md_t   md_handle,
                      ptl_size_t        local_offset   ,
                      ptl_size_t        length ,
                      ptl_process_id_t  target_id  ,
                      ptl_pt_index_t    pt_index ,
                      ptl_match_bits_t  match_bits ,
                      void              *user_ptr ,
                      ptl_size_t        remote_offset  ,
                      ptl_handle_ct_t   ct_handle ,
                      ptl_size_t          threshold );
```

Arguments

md_handle, *target_id*, *pt_index*, *match_bits*, *user_ptr*, *remote_offset*, *local_offset*, *length*	**input**	See the discussiton for **PtlGet()**.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.
PTL_CT_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.16.3 PtlTriggeredAtomic

The triggered atomic operations extend the Portals atomic operations (**PtlAtomic()**, **PtlFetchAtomic()**, and **PtlSwap()**) with the triggered operation semantics. When combined with triggered counting increments (**PtlTriggeredCTInc()**), triggered atomic operations enable an offloaded, non-blocking implementation of most collective operations.

Function Prototype for PtlTriggeredAtomic

```
int PtlTriggeredAtomic(ptl_handle_md_t     md_handle,
                       ptl_size_t          local_offset ,
                       ptl_size_t          length ,
                       ptl_ack_req_t       ack_req ,
                       ptl_process_id_t    target_id ,
                       ptl_pt_index_t      pt_index ,
                       ptl_match_bits_t    match_bits ,
                       ptl_size_t          remote_offset ,
                       void                *user_ptr ,
                       ptl_hdr_data_t      hdr_data,
                       ptl_op_t            operation ,
                       ptl_datatype_t      datatype ,
                       ptl_handle_ct_t     trig_ct_handle ,
                       ptl_size_t          threshold );
```

Arguments

md_handle, *local_offset*, *length*, *ack_req*, *target_id*, *pt_index*, *match_bits*, *remote_offset*, *user_ptr*, *hdr_data*, *operation*, *datatype*	**input**	See the discussion of **PtlAtomic()**.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *put_md_handle* or *get_md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.
PTL_CT_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.16.4 PtlTriggeredFetchAtomic

Function Prototype for PtlTriggeredFetchAtomic

```
int PtlTriggeredFetchAtomic(ptl_handle_md_t      get_md_handle,
                            ptl_size_t           local_get_offset ,
                            ptl_handle_md_t      put_md_handle,
                            ptl_size_t           local_put_offset ,
                            ptl_size_t           length ,
                            ptl_process_id_t     target_id ,
                            ptl_pt_index_t       pt_index ,
                            ptl_match_bits_t     match_bits ,
                            ptl_size_t           remote_offset ,
                            void                 *user_ptr ,
                            ptl_hdr_data_t       hdr_data,
                            ptl_op_t             operation ,
                            ptl_datatype_t       datatype ,
                            ptl_handle_ct_t      trig_ct_handle ,
                            ptl_size_t           threshold );
```

Arguments

get_md_handle, *local_get_offset*, *put_md_handle*, *local_put_offset*, *length*, *target_id*, *pt_index*, *match_bits*, *remote_offset*, *user_ptr*, *hdr_data*, *operation*, *datatype*	**input**	See the discussion of **PtlFetchAtomic()**.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *put_md_handle* or *get_md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.
PTL_CT_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.16.5 PtlTriggeredSwap

Function Prototype for PtlTriggeredSwap

```
int PtlTriggeredSwap(ptl_handle_md_t      get_md_handle,
                     ptl_size_t           local_get_offset ,
                     ptl_handle_md_t      put_md_handle,
                     ptl_size_t           local_put_offset ,
                     ptl_size_t           length ,
                     ptl_process_id_t     target_id ,
                     ptl_pt_index_t       pt_index ,
                     ptl_match_bits_t     match_bits ,
                     ptl_size_t           remote_offset ,
                     void                 *user_ptr ,
                     ptl_hdr_data_t       hdr_data,
                     void                 *operand,
                     ptl_op_t             operation ,
                     ptl_datatype_t       datatype ,
                     ptl_handle_ct_t      trig_ct_handle ,
                     ptl_size_t           threshold );
```

Arguments

get_md_handle, *local_get_offset*, *put_md_handle*, *local_put_offset*, *length*, *target_id*, *pt_index*, *match_bits*, *remote_offset*, *user_ptr*, *hdr_data*, *operand*, *operation*, *datatype*	**input**	See the discussion of **PtlSwap()**.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *put_md_handle* or *get_md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.
PTL_CT_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.16.6 PtlTriggeredCTInc

The triggered counting event increment extends the counting event increment (**PtlCTInc()**) with the triggered operation semantics. It is a convenient mechanism to provide chaining of dependencies between counting events. This allows a relatively arbitrary ordering of operations. For example, a **PtlTriggeredPut()** and a **PtlTriggeredCTInc()** could be dependent on *ct_handle* A with the same threshold. If the **PtlTriggeredCTInc()** is set to increment *ct_handle* B and a second **PtlTriggeredPut()** is dependent on *ct_handle* B, the second **PtlTriggeredPut()** will occur after the first.

Function Prototype for PtlTriggeredCTInc

```
int PtlTriggeredCTInc(ptl_handle_ct_t     ct_handle ,
                      ptl_size_t          increment,
                      ptl_handle_ct_t     trig_ct_handle  ,
                      ptl_size_t          threshold );
```

Arguments

ct_handle, *increment*	**input**	See the discussion of **PtlCTInc()**.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_CT_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.17 Operations on Handles

Handles are opaque data types. The only operation defined on them by the portals API is a comparison function.

3.17.1 PtlHandleIsEqual

The **PtlHandleIsEqual()** function compares two handles to determine if they represent the same object.

Function Prototype for PtlHandleIsEqual

```
PtlHandleIsEqual(ptl_handle_any_t    handle1,
                 ptl_handle_any_t    handle2);
```

Arguments

handle1, *handle2*	**input**	An object handle. Either of these handles is allowed to be the constant value, `PTL_INVALID_HANDLE`, which represents the value of an invalid handle.

Discussion: ***PtlHandleIsEqual()*** *does not check whether handle1 and handle2 are valid; only whether they are equal.*

Return Codes

PTL_OK	Indicates that the handles are equivalent.
PTL_FAIL	Indicates that the two handles are not equivalent.

3.18 Summary

We conclude this chapter by summarizing the names introduced by the portals API. We start with the data types introduced by the API. This is followed by a summary of the functions defined by the API which is followed by a summary of the function return codes. Finally, we conclude with a summary of the other constant values defined by the API.

Table 3.3 presents a summary of the types defined by the portals API. The first column in this table gives the type name, the second column gives a brief description of the type, the third column identifies the section where the type is defined, and the fourth column lists the functions that have arguments of this type.

Table 3.3. Portals Data Types: Data Types Defined by the Portals API.

Name	Meaning	Sec	Functions
ptl_ack_req_t	acknowledgment request types	3.15.2	PtlPut(), PtlAtomic(), PtlTriggeredPut(), PtlTriggeredAtomic()
ptl_ct_type_t	counting event type	3.14.2	PtlCTAlloc()
ptl_ct_event_t	counting event structure	3.14.2	PtlCTAlloc()
ptl_event_kind_t	event kind	3.13.1	PtlEQGet(), PtlEQWait(), PtlEQPoll()
ptl_initiator_event_t	event queue entry	3.13.4	PtlEQGet(), PtlEQWait(), PtlEQPoll()
ptl_initiator_event_t	initiator event information	3.13.4	PtlEQGet(), PtlEQWait(), PtlEQPoll()
ptl_target_event_t	target event information	3.13.4	PtlEQGet(), PtlEQWait(), PtlEQPoll()
ptl_handle_any_t	any object handles	3.2.2	PtlNIHandle(), PtlHandleIsEqual()
ptl_handle_eq_t	event queue handles	3.2.2	PtlEQAlloc(), PtlEQFree(), PtlEQGet(), PtlEQWait(), PtlEQPoll()
ptl_handle_md_t	memory descriptor handles	3.2.2	PtlMDRelease() , PtlMEAppend(), PtlPut(), PtlGet(), PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggeredPut(), PtlTriggeredGet(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()
ptl_handle_me_t	match list entry handles	3.2.2	PtlMEAppend(), PtlMEUnlink()
ptl_handle_ni_t	network interface handles	3.2.2	PtlNIInit(), PtlNIFini(), PtlNIStatus(), PtlEQAlloc()
ptl_hdr_data_t	user header data	3.15.2	PtlPut(), PtlGet(), PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggeredPut(), PtlTriggeredGet(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()
ptl_interface_t	network interface identifiers	3.2.5	PtlNIInit()
ptl_jid_t	job identifier	3.2.6	PtlGetJid()
ptl_list_t	type of list attached to a portal table entry	3.12.2	PtlMEAppend()
ptl_match_bits_t	match (and ignore) bits	3.2.4	PtlMEAppend(), PtlPut(), PtlGet(), PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggeredPut(), PtlTriggeredGet(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()
ptl_iovec_t	scatter/gather buffer descriptors	3.10.2	PtlMEAppend(), PtlMDBind(), PtlMDRelease()
ptl_md_t	memory descriptors	3.10.1	PtlMDRelease(), PtlMDBind()
ptl_me_t	match list entries	3.12.1	PtlMEAppend()
ptl_nid_t	node identifiers	3.2.6	PtlGetId()
ptl_ni_fail_t	network interface specific failures	3.13.3	PtlEQGet(), PtlEQWait(), PtlEQPoll()
ptl_ni_limits_t	implementation dependent limits	3.5.1	PtlNIInit()
ptl_pid_t	process identifier	3.2.6	PtlGetId()

continued on next page

continued from previous page

Name	Meaning	Sec	Functions
ptl_process_id_t	process identifiers	3.8.1	PtlGetId(), PtlMEAppend(), PtlPut(), PtlGet(), PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggeredPut(), PtlTriggeredGet(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()
ptl_pt_index_t	portal table indexes	3.2.3	PtlMEAppend(), PtlPTAlloc(), PtlPTFree(), PtlPTEnable(), PtlPTDisable(), PtlPut(), PtlGet(), PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggeredPut(), PtlTriggeredGet(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()
ptl_rank_t	rank within job	3.2.6	PtlGetId()
ptl_seq_t	event sequence number	3.13.4	PtlEQGet(), PtlEQWait(), PtlEQPoll()
ptl_size_t	sizes	3.2.1	PtlEQAlloc(), PtlPut(), PtlGet(), PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggeredPut(), PtlTriggeredGet(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()
ptl_sr_index_t	status register indexes	3.2.7	PtlNIStatus()
ptl_sr_value_t	status register values	3.2.7	PtlNIStatus()
ptl_time_t	time in milliseconds	3.13.9	PtlEQPoll()
ptl_uid_t	user identifier	3.2.6	PtlGetUid()

Table 3.4 presents a summary of the functions defined by the portals API. The first column in this table gives the name for the function, the second column gives a brief description of the operation implemented by the function, and the third column identifies the section where the function is defined.

Table 3.4. Portals Functions: Functions Defined by the Portals API.

Name	Operation	Definition
PtlCTAlloc()	create a counting event	3.14.2
PtlCTFree()	free a counting event	3.14.3
PtlCTInc()	increment a counting event by a certain value	3.14.7
PtlCTGet()	get the current value of a counting event	3.14.4
PtlCTWait()	wait for a counting event to reach a certain value	3.14.5
PtlCTSet()	set a counting event to a certain value	3.14.6
PtlEQAlloc()	create an event queue	3.13.5
PtlEQFree()	release the resources for an event queue	3.13.6
PtlEQGet()	get the next event from an event queue	3.13.7
PtlEQPoll()	poll for a new event on multiple event queues	3.13.9
PtlEQWait()	wait for a new event in an event queue	3.13.8
PtlFini()	shut down the portals API	3.4.2
PtlGet()	perform a *get* operation	3.15.3
PtlGetId()	get the identifier for the current process	3.8.2

continued on next page

continued from previous page

Name	Operation	Definition
PtlGetJid()	get the job identifier for the current process	3.9.1
PtlAtomic()	perform an atomic operation	3.15.5
PtlFetchAtomic()	perform an fetch and atomic operation	3.15.6
PtlSwap()	perform a swap operation	3.15.7
PtlGetUid()	get the network interface specific user identifier	3.7.1
PtlHandleIsEqual()	compares two handles to determine if they represent the same object	3.17.1
PtlInit()	initialize the portals API	3.4.1
PtlMDBind()	create a free-floating memory descriptor	3.10.3
PtlMDRelease()	release resources associated with a memory descriptor	3.10.4
PtlMEAppend()	create a match list entry and append it to a portal table	3.12.2
PtlMEUnlink()	remove a match list entry from a list and release its resources	3.12.3
PtlNIFini()	shut down a network interface	3.5.3
PtlNIHandle()	get the network interface handle for an object	3.5.5
PtlNIInit()	initialize a network interface	3.5.2
PtlNIStatus()	read a network interface status register	3.5.4
PtlPTAlloc()	allocate a free portal table entry	3.6.1
PtlPTFree()	free a portal table entry	3.6.2
PtlPTEnable()	enable a portal table entry that has been disabled	3.6.4
PtlPTDisable()	disable a portal table entry	3.6.3
PtlPut()	perform a *put* operation	3.15.2
PtlTriggeredAtomic()	perform a triggered atomic operation	3.16.3
PtlTriggeredFetchAtomic()	perform a triggered fetch and atomic operation	3.16.4
PtlTriggeredSwap()	perform a triggered swap operation	3.16.5
PtlTriggeredCTInc()	a triggered increment of a counting event by a certain value	3.16.6
PtlTriggeredGet()	perform a triggered *get* operation	3.16.2
PtlTriggeredPut()	perform a triggered *put* operation	3.16.1

Table 3.5 summarizes the return codes used by functions defined by the portals API. The first column of this table gives the symbolic name for the constant, the second column gives a brief description of the value, and the third column identifies the functions that can return this value.

Table 3.5. Portals Return Codes: Function Return Codes for the Portals API.

Name	Meaning	Functions
PTL_CT_INVALID	invalid counting event handle	**PtlCTFree()**, **PtlCTGet()**, **PtlCTWait()**
PTL_EQ_DROPPED	at least one event has been dropped	**PtlEQGet()**, **PtlEQWait()**
PTL_EQ_EMPTY	no events available in an event queue	**PtlEQGet()**
PTL_EQ_INVALID	invalid event queue handle	**PtlEQFree()**, **PtlEQGet()**
PTL_FAIL	error during initialization or cleanup	**PtlInit()**, **PtlFini()**
PTL_HANDLE_INVALID	invalid handle	**PtlNIHandle()**
PTL_IFACE_INVALID	initialization of an invalid interface	**PtlNIInit()**
PTL_MD_ILLEGAL	illegal memory descriptor values	**PtlMDRelease()**, **PtlMDBind()**
PTL_MD_IN_USE	memory descriptor has pending operations	**PtlMDRelease()**

continued on next page

continued from previous page

Name	Meaning	Functions
PTL_MD_INVALID	invalid memory descriptor handle	**PtlMDRelease()**
PTL_ME_IN_USE	ME has pending operations	**PtlMEUnlink()**
PTL_ME_INVALID	invalid match list entry handle	**PtlMEAppend()**
PTL_ME_LIST_TOO_LONG	match list entry list too long	**PtlMEAppend()**
PTL_NI_INVALID	invalid network interface handle	**PtlNIFini()**, **PtlMDBind()**, **PtlEQAlloc()**
PTL_NI_NOT_LOGICAL	not a logically addressed network interface handle	**PtlNIInit()**
PTL_NO_INIT	uninitialized API	*all*, except **PtlInit()**
PTL_NO_SPACE	insufficient memory	**PtlNIInit()**, **PtlMDBind()**, **PtlEQAlloc()**, **PtlMEAppend()**
PTL_OK	success	*all*
PTL_PID_INVALID	invalid pid	**PtlNIInit()**
PTL_PID_INUSE	pid is in use	**PtlNIInit()**
PTL_PROCESS_INVALID	invalid process identifier	**PtlNIInit()**, **PtlMEAppend()**, **PtlPut()**, **PtlGet()**
PTL_PT_FULL	portal table is full	**PtlPTAlloc()**
PTL_PT_EQ_NEEDED	EQ must be attached when flow control is enabled	**PtlPTAlloc()**
PTL_PT_INDEX_INVALID	invalid portal table index	**PtlMEAppend()**, **PtlPTFree()**
PTL_PT_IN_USE	portal table index is busy	**PtlPTFree()**
PTL_SEGV	addressing violation	**PtlNIInit()**, **PtlNIStatus()**, **PtlNIHandle()**, **PtlMDBind()**, **PtlEQAlloc()**, **PtlEQGet()**, **PtlEQWait()**
PTL_SR_INDEX_INVALID	invalid status register index	**PtlNIStatus()**

Table 3.6 summarizes the remaining constant values introduced by the portals API. The first column in this table presents the symbolic name for the constant, the second column gives a brief description of the value, the third column identifies the type for the value, and the fourth column identifies the sections in which the constant is mentioned. (A boldface section indicates the place the constant is introduced or described.)

Table 3.6. Portals Constants: Other Constants Defined by the Portals API.

Name	Meaning	Base Type	Reference
PTL_ACK_REQ	request an acknowledgment	**ptl_ack_req_t**	**3.15**, 3.15.2
PTL_CT_ACK_REQ	request a counting acknowledgment	**ptl_ack_req_t**	**3.15**, 3.15.2
PTL_OC_ACK_REQ	request an operation completed acknowledgment	**ptl_ack_req_t**	**3.15**, 3.15.2
PTL_CT_BYTE	a flag to indicate a counting event that counts bytes	**ptl_ct_type_t**	**3.14.2**
PTL_CT_NONE	a NULL count handle	**ptl_handle_ct_t**	**3.2.2**, 3.10.1
PTL_EQ_NONE	a NULL event queue handle	**ptl_handle_eq_t**	**3.2.2**, 3.10.1
PTL_EVENT_ACK	acknowledgment event	**ptl_event_kind_t**	**3.13.1**, 3.15.2

continued on next page

continued from previous page

Name	Meaning	Base Type	Reference
PTL_EVENT_GET	get event	**ptl_event_kind_t**	**3.13.1**, 3.15.3
PTL_EVENT_ATOMIC	atomic event	**ptl_event_kind_t**	**3.13.1**, 3.15.5
PTL_EVENT_DROPPED	overflow list exhaustion	**ptl_event_kind_t**	**3.13.1**
PTL_EVENT_PUT	put event	**ptl_event_kind_t**	**3.13.1**, 3.15.2
PTL_EVENT_PUT_OVERFLOW	put event overflow	**ptl_event_kind_t**	**3.13.1**, 3.15.2
PTL_EVENT_REPLY	reply event	**ptl_event_kind_t**	**3.13.1**, 3.15.3, 3.15.5
PTL_EVENT_SEND	send event	**ptl_event_kind_t**	**3.13.1**, 3.15.2, 3.15.5
PTL_EVENT_UNLINK	unlink event	**ptl_event_kind_t**	3.12.1, 3.12.3, **3.13.1**
PTL_EVENT_FREE	free event	**ptl_event_kind_t**	3.12.1, 3.12.3, **3.13.1**
PTL_EVENT_PT_DISABLED	portal table entry disabled event	**ptl_event_kind_t**	**3.13.1**, 3.12.1, 2.3
PTL_EVENT_PROBE	probe event	**ptl_event_kind_t**	3.12.1, 3.12.3, **3.13.1**
PTL_IFACE_DEFAULT	default interface	**ptl_interface_t**	**3.2.5**
PTL_INVALID_HANDLE	invalid handle	**ptl_handle_any_t**	**3.2.2**, 3.17.1
PTL_JID_ANY	wildcard for job identifier	**ptl_jid_t**	3.9, **3.2.6**, 3.11, 3.12
PTL_JID_NONE	job identifiers not supported for process	**ptl_jid_t**	**3.9**
PTL_PRIORITY_LIST	specifies the priority list attached to a portal table entry	int	**3.12.2**
PTL_MD_EVENT_DISABLE	a flag to disable events	int	**3.10.1**
PTL_MD_EVENT_SUCCESS_DISABLE	a flag to disable events that indicate success	int	**3.10.1**
PTL_LE_ACK_DISABLE	a flag to disable acknowledgments	int	**3.11.1**
PTL_LE_AUTH_USE_JID	a flag to indicate that the job ID should be used for access control	int	**3.11.1**
PTL_LE_EVENT_DISABLE	a flag to disable events	int	**3.11.1**
PTL_LE_EVENT_SUCCESS_DISABLE	a flag to disable events that indicate success	int	**3.11.1**
PTL_LE_EVENT_CT_GET	a flag to count get events	int	**3.11.1**
PTL_LE_EVENT_CT_PUT	a flag to count put events	int	**3.11.1**
PTL_LE_EVENT_CT_PUT_OVERFLOW	a flag to count “overflow” put events	int	**3.11.1**
PTL_LE_EVENT_CT_ATOMIC	a flag to count atomic events	int	**3.11.1**
PTL_LE_EVENT_CT_ATOMIC_OVERFLOW	a flag to count “overflow” atomic events	int	**3.11.1**
PTL_LE_EVENT_UNLINK_DISABLE	a flag to disable unlink events	int	**3.11.1**
PTL_LE_OP_GET	a flag to enable *get* operations	int	**3.11.1**, 4.2
PTL_LE_OP_PUT	a flag to enable *put* operations	int	**3.11.1**, 4.2

continued on next page

continued from previous page

Name	Meaning	Base Type	Reference
PTL_LE_USE_ONCE	a flag to indicate that the list entry will only be used once	int	**3.11.1**
PTL_LE_MAY_ALIGN	a flag to indicate that the implementation may align an incoming message to a natural boundary to enhance performance	int	**3.11.1**
PTL_ME_ACK_DISABLE	a flag to disable acknowledgments	int	**3.12.1**
PTL_ME_AUTH_USE_JID	a flag to indicate that the job ID should be used for access control	int	**3.12.1**
PTL_ME_EVENT_DISABLE	a flag to disable events	int	**3.12.1**
PTL_ME_EVENT_SUCCESS_DISABLE	a flag to disable events that indicate success	int	**3.12.1**
PTL_ME_EVENT_CT_GET	a flag to count get events	int	**3.12.1**
PTL_ME_EVENT_CT_PUT	a flag to count put events	int	**3.12.1**
PTL_ME_EVENT_CT_PUT_OVERFLOW	a flag to count “overflow” put events	int	**3.12.1**
PTL_ME_EVENT_CT_ATOMIC	a flag to count atomic events	int	**3.12.1**
PTL_ME_EVENT_CT_ATOMIC_OVERFLOW	a flag to count “overflow” atomic events	int	**3.12.1**
PTL_MD_EVENT_CT_SEND	a flag to count send events	int	**3.10.1**
PTL_MD_EVENT_CT_REPLY	a flag to count reply events	int	**3.10.1**
PTL_MD_EVENT_CT_ACK	a flag to count acknowledgment events	int	**3.10.1**
PTL_MD_UNORDERED	a flag to indicate that messages from this MD do not need to be ordered	int	**3.10.1**
PTL_MD_REMOTE_FAILURE_DISABLE	a flag to indicate that remote failures should not be delivered to the local EQ	int	**3.10.1**
PTL_IOVEC	a flag to enable scatter/gather memory descriptors	int	**3.12.1**, 3.10.2
PTL_ME_EVENT_UNLINK_DISABLE	a flag to disable unlink events	int	**3.12.1**
PTL_ME_MANAGE_LOCAL	a flag to enable the use of local offsets	int	**3.12.1**, 3.15.2, 3.15.3
PTL_ME_MIN_FREE	use the *min_free* field in a match list entry	unsigned int	**3.12.1**
PTL_ME_OP_GET	a flag to enable *get* operations	int	**3.12.1**, 4.2
PTL_ME_OP_PUT	a flag to enable *put* operations	int	**3.12.1**, 4.2
PTL_ME_NO_TRUNCATE	a flag to disable truncation of a request	int	**3.12.1**, 4.2

continued on next page

continued from previous page

Name	Meaning	Base Type	Reference
PTL_ME_USE_ONCE	a flag to indicate that the match list entry will only be used once	int	**3.12.1**
PTL_ME_MAY_ALIGN	a flag to indicate that the implementation may align an incoming message to a natural boundary to enhance performance	int	**3.12.1**
PTL_NID_ANY	wildcard for node identifier fields	**ptl_nid_t**	**3.2.6**, 3.12.2, 3.12
PTL_NI_OK	successful event	**ptl_ni_fail_t**	**3.13.3**, 3.13.4
PTL_NI_UNDELIVERABLE	message could not be delivered	**ptl_ni_fail_t**	**3.13.3**, 3.13.4
PTL_NI_FLOW_CTRL	message encounterd a flow control condition	**ptl_ni_fail_t**	**3.13.3**, 3.13.4
PTL_NI_PERM_VIOLATION	message encounterd a permissions violation	**ptl_ni_fail_t**	**3.13.3**, 3.13.4
PTL_NI_MATCHING	a flag to indicate that the network interface must provide matching portals addressing	int	**3.5.2**
PTL_NI_NO_MATCHING	a flag to indicate that the network interface must provide non-matching portals addressing	int	**3.5.2**
PTL_NI_LOGICAL	a flag to indicate that the network interface must provide logical addresses for network end-points	int	**3.5.2**
PTL_NI_PHYSICAL	a flag to indicate that the network interface must provide physical addresses for network end-points	int	**3.5.2**
PTL_NO_ACK_REQ	request no acknowledgment	**ptl_ack_req_t**	**3.15**, 3.15.2, 4.1
PTL_CT_OPERATION	a flag to indicate a counting event that counts operations	**ptl_ct_type_t**	**3.14.2**
PTL_OVERFLOW	specifies the overflow list attached to a portal table entry	int	**3.12.2**
PTL_PID_ANY	wildcard for process identifier fields	**ptl_pid_t**	**3.2.6**, 3.5.2, 3.12.2, 3.12
PTL_PT_ANY	wildcard for portal table entry identifier fields	**ptl_pt_index_t**	**3.6.1**
PTL_PT_ONLY_USE_ONCE	a flag to indicate that the portal table entry will only have entries with the PTL_ME_USE_ONCE or PTL_LE_USE_ONCE option set	int	**3.6.1**

continued on next page

continued from previous page

Name	Meaning	Base Type	Reference
PTL_PROBE_ONLY	specifies that the match list entry should not be attached, but should probe only	int	**3.12.2**
PTL_RANK_ANY	wildcard for rank fields	**ptl_rank_t**	**3.2.6**, 3.12.2, 3.12
PTL_SR_DROP_COUNT	index for the dropped count register	**ptl_sr_index_t**	**3.2.7**, 3.5.4
PTL_SR_PERMISSIONS_VIOLATIONS	index for the permission violations register	**ptl_sr_index_t**	**3.2.7**, 3.5.4
PTL_TIME_FOREVER	a flag to indicate unbounded time	**ptl_time_t**	**3.13.9**
PTL_UID_ANY	wildcard for user identifier	**ptl_uid_t**	**3.2.6**, 3.12.2, 3.11, 3.12

Chapter 4

The Semantics of Message Transmission

The portals API uses five types of messages: *put*, *acknowledgment*, *get*, *reply*, and *atomic*. In this section, we describe the information passed on the wire for each type of message. We also describe how this information is used to process incoming messages.

4.1 Sending Messages

Table 4.1 summarizes the information that is transmitted for a *put* request. The first column provides a descriptive name for the information, the second column provides the type for this information, the third column identifies the source of the information, the fourth column provides an approximate size for the item, and the fourth column provides additional notes. Most information that is transmitted is obtained directly from the *put* operation.

IMPLEMENTATION NOTE 34:

Information on the wire

This section describes the information that portals semantics require to be passed between an *initiator* and its *target*. The portals specification does not enforce a given wire protocol or in what order and what manner information is passed along the communication path.

For example, portals semantics require that an *acknowledgment* event contains the *user_ptr* and it must be placed in the event queue referenced by the *eq_handle* found in the MD referenced by the *md_handle* associated with the *put*; i.e., the *acknowledgment* event provides a pointer that the application can use to identify the operation and must be placed the in the right memory descriptor's event queue. One approach would be to send the *user_ptr* and *md_handle* to the *target* in the *put* and back again in the *acknowledgment* message. If an implementation has another way of tracking the *user_ptr* and *md_handle* at the initiator, then sending the *user_ptr* and *md_handle* should not be necessary.

Notice that the *match_bits*, *md_handle* and *user_ptr* fields in the *put* operation are optional. If the *put* is originating from a non-matching network interface, there is no need for the *match_bits* to be transmitted since the destination will ignore them. Similarily, if no acknowledgement was requested, *md_handle* and *user_ptr* do not need to be sent. If an acknowlegement is requested (either `PTL_CT_ACK_REQ`, `PTL_ACK_REQ`, or `PTL_OC_ACK_REQ`), then the *md_handle* may be sent in the *put* message so that the *target* can send it back to the *initiator* in the *acknowledgment* message. The *md_handle* is needed by the *initiator* to find the right event queue for the acknowlegement event. The *user_ptr* is only required in the case of a full acknowlegment (`PTL_ACK_REQ`). `PTL_CT_ACK_REQ` and `PTL_OC_ACK_REQ` requests do not require the *user_ptr* field to generate the acknowlegment event at the *initiator* of the *put* operation.

A portals header contains 8 bytes of user supplied data specified by the *hdr_data* argument passed to **PtlPut()**. This is useful for out-of-band data transmissions with or without bulk data. The header bytes are stored in the event generated at the *target*. (See Section 3.15.2 on page 82.)

IMPLEMENTATION NOTE 35:

Size of data on the wire

Table 4.1 specifies sizes for each data item that are conformant to the Portals 4.0 specification; however, a given implementation can impose additional constraints to reduce the size of some of these fields. For example, the *remote_offset* could each be reduced to 5 bytes on a platform that supported less than 1 TB of memory. Further reductions for the special case of the non-matching operation with only a PTL_CT_ACK_REQ or PTL_OC_ACK_REQ would reduce the Portals Send Request significantly. Similar optimizations are available in other pieces of wire information.

Table 4.1. Send Request: Information Passed in a Send Request — **PtlPut()**.

Information	Type	PtlPut() Argument	Size	Notes
operation	int		4b	indicates a *put* request
ack type	**ptl_ack_req_t**	*ack_req*	2b	
options	unsigned int	*md_handle*	2b	*options* field from NI associated with MD
job identifier	**ptl_jid_t**		4B	local information (if supported)
initiator	**ptl_process_id_t**		4B	local information
user	**ptl_uid_t**		4B	local information
target	**ptl_process_id_t**	*target_id*	4B	
portal index	**ptl_pt_index_t**	*pt_index*	1B	
match bits	**ptl_match_bits_t**	*match_bits*	8B	opt. if *options*.PTL_NI_NO_MATCHING
offset	**ptl_size_t**	*remote_offset*	8B	
memory desc	**ptl_handle_md_t**	*md_handle*	2B	opt. if *ack_req* =PTL_NO_ACK_REQ
header data	**ptl_hdr_data_t**	*hdr_data*	8B	user data in header
put user pointer	void *	*user_ptr*	8B	opt. if *ack_req* =PTL_NO_ACK_REQ or *ack_req* =PTL_CT_ACK_REQ or *ack_req* =PTL_OC_ACK_REQ
length	**ptl_size_t**	*length*	8B	*length* argument
data	*bytes*	*md_handle*		user data
total	unsigned int		61B	

Tables 4.2 and 4.3 summarizes the information transmitted in an *acknowledgment*. Most of the information is simply echoed from the *put* request. Notice that the *initiator* and *target* are obtained directly from the *put* request but are swapped in generating the *acknowledgment*. The only new pieces of information in the *acknowledgment* are the manipulated length, which is determined as the *put* request is satisfied, and the actual offset used.

IMPLEMENTATION NOTE 36: Acknowledgment requests

If an *acknowledgment* has been requested, the associated memory descriptor remains in use by the implementation until the *acknowledgment* arrives and can be logged in the event queue. See Section 3.10.4 for how pending operations affect unlinking of memory descriptors.

If the target memory descriptor has the `PTL_ME_MANAGE_LOCAL` flag set, the offset local to the *target* memory descriptor is used. If the flag is set, the offset requested by the *initiator* is used. An *acknowledgment* message returns the actual value used.

Lightweight "counting" acknowlegments do not require the actual offset used or user pointer since they do not generate a **ptl_initiator_event_t** at the *put* operation *initiator*.

Table 4.2. Acknowledgment: Information Passed in an Acknowledgment.

Information	Type	PtlPut() Argument	Size	Notes
operation	int		4b	indicates an *acknowledgment*
options	unsigned int	*put_md_handle*	2b	*options* field from NI associated with MD
initiator	**ptl_process_id_t**	*target_id*	4B	echo *target* of *put*
target	**ptl_process_id_t**	*initiator*	4B	echo *initiator* of *put*
memory descriptor	**ptl_handle_md_t**	*md_handle*	2B	echo *md_handle* of *put*
put user pointer	void *	*user_ptr*	8B	echo *user_ptr* of *put*
offset	**ptl_size_t**	*remote_offset*	8B	obtained from the operation
manipulated length	**ptl_size_t**		8B	obtained from the operation
Total	unsigned int		35B	

Table 4.3. Acknowledgment: Information Passed in a "Counting" Acknowledgment.

Information	Type	PtlPut() Argument	Size	Notes
operation	int		4b	indicates an *acknowledgment*
options	unsigned int	*put_md_handle*	2b	*options* field from NI associated with MD
initiator	**ptl_process_id_t**	*target_id*	4B	local information on *put* target
target	**ptl_process_id_t**	*initiator*	4B	echo *initiator* of *put*
memory descriptor	**ptl_handle_md_t**	*md_handle*	2B	echo *md_handle* of *put*
manipulated length	**ptl_size_t**		8B	obtained from the operation
Total	unsigned int		19B	

Table 4.4 summarizes the information that is transmitted for a *get* request. Like the information transmitted in a *put* request, most of the information transmitted in a *get* request is obtained directly from the **PtlGet()** operation. The memory descriptor must not be unlinked until the *reply* is received.

Table 4.5 summarizes the information transmitted in a *reply*. Like an *acknowledgment*, most of the information is simply echoed from the *get* request. The *initiator* and *target* are obtained directly from the *get* request but are swapped in generating the *reply*. The only new information in the *reply* are the manipulated length, the actual offset used, and the data, which are determined as the *get* request is satisfied.

Table 4.4. Get Request: Information Passed in a Get Request — **PtlGet()** and **PtlGetRegion()**.

Information	Type	PtlGet() Argument	Size	Notes
operation	int		4b	indicates a *get* operation
options	unsigned int	*md_handle*	2b	*options* field from NI associated with MD
job identifier	**ptl_jid_t**		4B	local information (if supported)
initiator	**ptl_process_id_t**		4B	local information
user	**ptl_uid_t**		4B	local information
target	**ptl_process_id_t**	*target_id*	4B	
portal index	**ptl_pt_index_t**	*pt_index*	1B	
match bits	**ptl_match_bits_t**	*match_bits*	8B	optional if the PTL_NI_NO_MATCHING option is set.
offset	**ptl_size_t**	*remote_offset*	8B	
memory descriptor	**ptl_handle_md_t**	*md_handle*	2B	destination of *reply*
length	**ptl_size_t**	*length*	8B	
initiator offset	**ptl_size_t**	*local_offset*	8B	
get user pointer	void *	*user_ptr*	8B	
Total	unsigned int		61B	

Table 4.5. Reply: Information Passed in a Reply.

Information	Type	PtlGet() Argument	Size	Notes
operation	int		4b	indicates an *reply*
options	unsigned int	*get_md_handle*	2b	*options* field from NI associated with MD
initiator	**ptl_process_id_t**	*target_id*	4B	local information on *get* target
target	**ptl_process_id_t**	*initiator*	4B	echo *initiator* of *get*
memory descriptor	**ptl_handle_md_t**	*md_handle*	2B	echo *md_handle* of *get*
initiator offset	**ptl_size_t**	*local_offset*	8B	echo *local_offset* of *get*
get user pointer	void *	*user_ptr*	8B	echo *user_ptr* of *get*
manipulated length	**ptl_size_t**		8B	obtained from the operation
offset	**ptl_size_t**	*remote_offset*	8B	obtained from the operation
data	*bytes*			obtained from the operation
Total	unsigned int		43B	

Table 4.6 presents the information that needs to be transmitted from the *initiator* to the *target* for an *atomic* operation. The result of an *atomic* operation is a *reply* and (optionally) an *acknowledgment* as described in Table 4.5.

4.2 Receiving Messages

When an incoming message arrives on a network interface, the communication system first checks that the *target* process identified in the request is a valid process that has initialized the network interface (i.e., that the *target* process has a valid portal table). If this test fails, the communication system discards the message and increments the dropped message count for the interface. The remainder of the processing depends on the type of the incoming message. *put*, *get*, and *atomic* messages go through portals address translation (searching a list) and must then pass an access control test. In contrast, *acknowledgment* and *reply* messages bypass the access control checks and the translation step.

Table 4.6. Atomic Request: Information Passed in an Atomic Request.

Information	Type	PtlAtomic() Argument	Size	Notes
operation	int		2B	indicates the type of *atomic* operation and datatype
options	unsigned int	*put_md_handle*	2b	*options* field from NI associated with MD
ack type	**ptl_ack_req_t**	*ack_req*	2b	
job identifier	**ptl_jid_t**		4B	local information (if supported)
initiator	**ptl_process_id_t**		4B	local information
user	**ptl_uid_t**		4B	local information
target	**ptl_process_id_t**	*target_id*	4B	
portal index	**ptl_pt_index_t**	*pt_index*	1B	
memory descriptor	**ptl_handle_md_t**	*put_md_handle*	2B	opt. if *ack_req* =PTL_NO_ACK_REQ
user pointer	void *	*user_ptr*	8B	opt. if *ack_req* =PTL_NO_ACK_REQ or *ack_req* =PTL_CT_ACK_REQ or *ack_req* =PTL_OC_ACK_REQ
match bits	**ptl_match_bits_t**	*match_bits*	8B	optional if the PTL_NI_NO_MATCHING option is set.
offset	**ptl_size_t**	*remote_offset*	8B	
memory descriptor	**ptl_handle_md_t**	*get_md_handle*	2B	destination of *reply*
length	**ptl_size_t**	*put_md_handle*	8B	*length* member
operand	*bytes*	operand	8B	Used in CSWAP and MSWAP operations
data	*bytes*	*put_md_handle*		user data
Total	unsigned int		65B	

Acknowledgment messages include the memory descriptor handle used in the original **PtlPut()** operation. This memory descriptor will identify the event queue where the event should be recorded. Upon receipt of an acknowledgment, the runtime system only needs to confirm that the memory descriptor and event queue still exist. Should any of these conditions fail, the message is simply discarded, and the dropped message count for the interface is incremented. Otherwise, the system builds an acknowledgment event from the information in the acknowledgment message and adds it to the event queue.

Reception of *reply* messages is also relatively straightforward. Each *reply* message includes a memory descriptor handle. If this descriptor exists, it is used to receive the message. A *reply* message will be dropped if the memory descriptor identified in the request does not exist or it has become inactive. In this case, the dropped message count for the interface is incremented. Every memory descriptor accepts and truncates incoming *reply* messages, eliminating the other potential reasons for rejecting a *reply* message.

The critical step in processing an incoming *put*, *get*, or *atomic* request involves mapping the request to a match list entry (or list entry). This step starts by using the portal index in the incoming request to identify a list of match list entries (or list entries). On a matching interface, the list of match list entries is searched in sequential order until a match list entry is found whose match criteria matches the match bits in the incoming request and that accepts the request. On a non-matching interface, the first item on the list is used and a permissions check is performed.

Because *acknowledgment* and *reply* messages are generated in response to requests made by the process receiving these messages, the checks performed by the runtime system for acknowledgments and replies are minimal. In contrast, *put*, *get*, and *atomic* messages are generated by remote processes and the checks performed for these messages are more extensive. Incoming *put*, *get*, or *atomic* messages may be rejected because:

- the portal index supplied in the request is not valid;
- the match bits supplied in the request do not match any of the match list entries that accepts the request, or
- the access control information provided in the list entry does not match the information provided in the message.

In all cases, if the message is rejected, the incoming message is discarded and the dropped message count for the interface is incremented.

A list entry or match list entry may reject an incoming request if the PTL_ME_OP_PUT or PTL_ME_OP_GET option has not been enabled and the operation is *put*, *get*, or *atomic* (Table 4.7). In addition, a match list entry may reject an incoming request if the length specified in the request is too long for the match list entry and the PTL_ME_NO_TRUNCATE option has been enabled. Truncation is always enabled on standard list entries; thus, a message cannot be rejected for this reason on a non-matching NI.

Also see Sections 2.2 and Figure 2.9.

Table 4.7. Portals Operations and ME/LE Flags: A - indicates that the operation will be rejected, and a • indicates that the operation will be accepted.

Target ME/LE Flags	Operation		
	put	*get*	*atomic*
none	-	-	-
PTL_ME_OP_PUT/PTL_LE_OP_PUT	•	-	-
PTL_ME_OP_GET/PTL_LE_OP_GET	-	•	-
both	•	•	•

References

Alverson, R. (2003, August). **Red Storm**. In *Invited Talk, Hot Chips 15*.

Brightwell, R., D. S. Greenberg, A. B. Maccabe, and R. Riesen (2000, February). **Massively Parallel Computing with Commodity Components**. *Parallel Computing 26*, 243–266.

Brightwell, R., T. Hudson, K. T. Pedretti, and K. D. Underwood (2006, May/June). **SeaStar Interconnect: Balanced Bandwidth for Scalable Performance**. *IEEE Micro 26*(3).

Brightwell, R., T. Hudson, R. Riesen, and A. B. Maccabe (1999, December). **The Portals 3.0 Message Passing Interface**. Technical Report SAND99-2959, Sandia National Laboratories.

Brightwell, R. and L. Shuler (1996, July). **Design and Implementation of MPI on Puma Portals**. In *Proceedings of the Second MPI Developer's Conference*, pp. 18–25.

Compaq, Microsoft, and Intel (1997, December). **Virtual Interface Architecture Specification Version 1.0**. Technical report, Compaq, Microsoft, and Intel.

Cray Research, Inc. (1994, October). **SHMEM Technical Note for C, SG-2516 2.3**. Cray Research, Inc.

Infiniband Trade Association (1999). **http://www.infinibandta.org**.

Ishikawa, Y., H. Tezuka, and A. Hori (1996). **PM: A High-Performance Communication Library for Multi-user Parallel Envrionments**. Technical Report TR-96015, RWCP.

Lauria, M., S. Pakin, and A. Chien (1998). **Efficient Layering for High Speed Communication: Fast Messages 2.x**. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*.

Maccabe, A. B., K. S. McCurley, R. Riesen, and S. R. Wheat (1994, June). **SUNMOS for the Intel Paragon: A Brief User's Guide**. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.*, pp. 245–251.

Message Passing Interface Forum (1994). **MPI: A Message-Passing Interface standard**. *The International Journal of Supercomputer Applications and High Performance Computing 8*, 159–416.

Message Passing Interface Forum (1997, July). **MPI-2: Extensions to the Message-Passing Interface**. Message Passing Interface Forum.

Myricom, Inc. (1997). **The GM Message Passing System**. Technical report, Myricom, Inc.

Riesen, R., R. Brightwell, and A. B. Maccabe (2005). **The Evolution of Portals, an API for High Performance Communication**. *To be published*.

Riesen, R., R. Brightwell, A. B. Maccabe, T. Hudson, and K. Pedretti (2006, January). **The Portals 3.3 Message Passing Interface: Document Revision 2.0**. Technical report SAND2006-0420, Sandia National Laboratories.

NOTE: This is the final version of the document for Portals version 3.3. It supersedes SAND99-2959

Shuler, L., C. Jong, R. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup (1995). **The Puma Operating System for Massively Parallel Computers**. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group.

Task Group of Technical Committee T11 (1998, July). **Information Technology - Scheduled Transfer Protocol - Working Draft 2.0**. Technical report, Accredited Standards Committee NCITS.

Appendix A

Frequently Asked Questions

This document is a specification for the portals 4.0 API. People using and implementing Portals sometimes have questions that the specification does not address. In this appendix we answer some of the more common questions.

Q Are Portals a wire protocol?

A No. The portals document defines an API with semantics that specify how messages move from one address space to another. It does not specify how the individual bytes are transferred. In that sense it is similar to the socket API: TCP/IP or some other protocol is used to reliably transfer the data. Portals assume an underlying transport mechanism that is reliable and scalable.

Q How are Portals different from the sockets API (TCP/IP) or the MPI API?

A Sockets are stream-based while Portals are message-based. Portals implementations can use the a priori knowledge of the total message length to manage the buffers and protocols to be used. The portals API makes it easy to let the implementation know in advance where in user space incoming data should be deposited. The sockets API makes this more difficult because the implementation will not know where data has to go until the application issues a *read()* request.

The sockets API using TCP/IP is connection-oriented which limits scalability because state has to be maintained for each open connection and the number of connections increases with the size of the machine.

MPI is a higher level API than Portals. In many ways, it provides simpler semantics and APIs. It also provides a variety of higher level APIs (derived data types, collective operations) that Portals does not.

Portals are ideally suited to be used by an MPI implementation. An application programmer, however, may grow frustrated by Portals' lack of user-friendliness. We recommend that Portals be used by systems programmers and library writers, not application programmers.

Q What about GM, FM, AM, PM, etc.?

A There are many communication paradigms, and, especially in the early 1990s, many experiments were conducted on how to best pass messages among supercomputer nodes; hence, the proliferation of the various *M message passing layers.

Some of them, such as GM, are hardware specific. Almost every network interface vendor has its own API to access its hardware. Portals are portable and open source. They were designed to run on a wide variety of networks with NICs that are programmable or not. This was an important design criteria for Portals 3.0 when work on Cplant started.

Most of the research message passing layers do not provide reliability because they were designed for networks that are, for all practical purposes, reliable. While Portals themselves do not provide a wire protocol, Portals demand that the transport layer underneath is reliable. This places Portals a level above the other APIs in the networking stack. On

reliable networks, such as ASCI Red, Portals can be implemented without a wire protocol. On unreliable networks, such as Myrinet, Portals can run over GM or some other protocol that implements reliability.

Some of the research paradigms do not scale to thousands of nodes. In order to control local resources, some of them use send tokens to limit the number of messages that can be sent through the network at any given time. As a machine and its network grow, this imposes severe limitations and degrades the scalability of the message passing layer.

Q What is a NAL?

A NAL stands for Network Abstraction Layer. All current portals 3.x implementations are in some way or another derived from the reference implementation which employs a NAL. A NAL is a very nice way to abstract the network interface from a portals library. The library implements common portals functions in user space and can be easily ported from one architecture to another. On the other side of the NAL, in protected space, we find routines that are more specific to a given architecture and network interface.

Q Must Portals be implemented using a NAL?

A No. A NAL provides a nice abstraction and makes it easier to port portals implementations, but the API and semantics of Portals do not require a NAL.

Q Why does the portals API not specify a barrier operation?

A Earlier versions of the API had a barrier function. It turned out to be quite difficult to implement on some architectures. The main problem was that nodes would boot in intervals and not be ready to participate in a portals barrier operation until later. The portals implementations had to rely on the runtime system to learn when nodes became active. The runtime systems, in turn, usually had some form of barrier operation that allowed them to synchronize nodes after booting or after job load.

Because that functionality already existed and it made portals implementations difficult, we decided to eliminate the barrier operation from the portals API. However, future versions of Portals may have collective operations. In that case, the portals barrier would be re-introduced.

Appendix B

Portals Design Guidelines

Early versions of Portals were based on the idea to use data structures to describe to the transport mechanism how data should be delivered. This worked well for the Puma OS on the Intel Paragon but not so well under Linux on Cplant. The solution was to create a thin API over those data structures and add a level of abstraction. The result was Portals 3.x. While Portals 3.x supported MPI well for kernel level implementations, more advanced offloading network interfaces and the rising importance of PGAS models exposed several weaknesses. This led to several enhancements that became Portals 4.x.

When designing and expanding this API, we were guided by several principles and requirements. We have divided them into three categories: requirements that must be fulfilled by the API and its implementations, requirements that should be met, and a wish list of things that would be nice if Portals 4.x could provide them.

B.1 Mandatory Requirements

Message passing protocols. Portals *must* support efficient implementations of commonly used message passing protocols.

Partitioned Global Address Space (PGAS) Support. Portals *must* support efficient implementations of typical PGAS languages and programming interfaces.

Portability. It *must* be possible to develop implementations of Portals on a variety of existing message passing interfaces.

Scalability. It *must* be possible to write efficient implementations of Portals for systems with thousands of nodes.

Performance. It *must* be possible to write high performance (e.g., low latency, high bandwidth) implementations of Portals on existing hardware and on hardware capable of offloading Portals processing.

Multiprocess support. Portals *must* support use of the communication interface by tens of processes per node.

Communication between processes from different executables. Portals *must* support the ability to pass messages between processes instantiated from different executables.

Runtime independence. The ability of a process to perform message passing *must not* depend on the existence of an external runtime environment, scheduling mechanism, or other special utilities outside of normal UNIX process startup.

Memory protection. Portals *must* ensure that a process cannot access the memory of another process without consent.

B.2 The *Will* Requirements

Operational API. Portals *will* be defined by operations, not modifications to data structures. This means that the interface will have explicit operations to send and receive messages. (It does not mean that the receive operation will involve a copy of the message body.)

MPI. It *will* be possible to write an efficient implementation of the point-to-point operations in MPI 1 using Portals.

PGAS. It *will* be possible to write an efficient implementation of the one-sided and atomic operations found in PGAS models using Portals.

Network Interfaces. It *will* be possible to write an efficient implementation of Portals using a network interface that provides offload support.

Operating Systems. It *will* be possible to write an efficient implementation of Portals using a lightweight kernel *or* Linux as the host OS.

Message Size. Portals *will not* impose an arbitrary restriction on the size of message that can be sent.

OS bypass. Portals *will* support an OS bypass message passing strategy. That is, high performance implementations of the message passing mechanisms will be able to bypass the OS and deliver messages directly to the application.

Put/Get. Portals *will* support remote put/get operations.

Packets. It *will* be possible to write efficient implementations of Portals that packetize message transmission.

Receive operation. The receive operation of Portals *will* use an address and length pair to specify where the message body should be placed.

Receiver managed communication. Portals *will* support receive-side management of message space, and this management will be performed during message receipt.

Sender managed communication. Portals *will* support send-side management of message space.

Parallel I/O. Portals *will* be able to serve as the transport mechanism for a parallel file I/O system.

Gateways. It *will* be possible to write *gateway* processes using Portals. A gateway process is a process that receives messages from one implementation of Portals and transmits them to another implementation of Portals.

Asynchronous operations. Portals *will* support asynchronous operations to allow computation and communication to overlap.

Receive side matching. Portals *will* allow matching on the receive side before data is delivered into the user buffer.

B.3 The *Should* Requirements

Message Alignment. Portals *should* not impose any restrictions regarding the alignment of the address(es) used to specify the contents of a message.

Striping. Portals *should* be able to take advantage of multiple interfaces on a single logical network to improve the bandwidth

Socket API. Portals *should* support an efficient implementation of sockets (including UDP and TCP/IP).

Scheduled Transfer. It *should* be possible to write an efficient implementation of Portals based on Scheduled Transfer (ST).

Virtual Interface Architecture. It *should* be possible to write an efficient implementation of Portals based on the Virtual Interface Architecture (VIA).

Internetwork consistency. Portals *should not* impose any consistency requirements across multiple networks/interfaces. In particular, there will not be any memory consistency/coherency requirements when messages arrive on independent paths.

Ease of use. Programming with Portals *should* be no more complex than programming traditional message passing environments such as UNIX sockets or MPI. An in-depth understanding of the implementation or access to implementation-level information should not be required.

Minimal API. Only the smallest number of functions and definitions necessary to manipulate the data structures should be specified. That means, for example, that convenience functions, which can be implemented with the already defined functions, will not become part of the API.

One exception to this is if a non-native implementation would suffer in scalability or take a large performance penalty.

Appendix C

A README Template

Each portals implementation should provide a README file that details implementation-specific choices. This appendix serves as a template for such a file by listing which parameters should be specified.

Limits. The call **PtlNIInit()** accepts a desired set of limits and returns a set of actual limits. The README should state the possible ranges of actual limits for this implementation, as well as the acceptable ranges for the values passed into **PtlNIInit()**. See Section 3.5.1

Status Registers. Portals define a set of status registers (Section 3.2.7). The type **ptl_sr_index_t** defines the mandatory `PTL_SR_DROP_COUNT` and `PTL_SR_PERMISSIONS_VIOLATIONS`, as well as all other, implementation specific indexes. The README should list what indexes are available and what their purposes are.

Network interfaces. Each portals implementation defines `PTL_IFACE_DEFAULT` to access the default network interface on a system (Sections 3.2.5 and 3.5.2). An implementation that supports multiple interfaces must specify the constants used to access the various interfaces through **PtlNIInit()**.

Portal table. The portals specification says that a compliant implementation must provide at least 8 entries per portal table (Section 3.5). The README file should state how many entries will actually be provided.

Job identifiers. The README file should indicate whether the implementation supports job identifiers (Section 3.9).

Alignment. If an implementation favors specific alignments for memory descriptors, the README should state what they are and the (performance) consequences if they are not observed (Sections 3.10.1 and 3.12.1).

Appendix D

Implementations

IMPLEMENTATION NOTE 37:	Implementations of Portals 3.3 This section describes implementations of Portals 3.3 in lieu of a new implementation of Portals 4.0. Note that the text is taken from the Portals 3.3 document and occasionally references that document. Many implementation concepts remain the same between the two versions.

In this appendix we briefly mention two portals 3.3 implementations: A reference implementation and one that runs on Cray's XT3/XT4/XT5 Red Storm.

D.1 Reference Implementation

A portals 3.3 reference implementation has been written and is maintained by Jim Schutt. The main goal of the reference implementation is to provide a working example that implements the syntax, semantics, and spirit of Portals as described in the version 3.3 document. While many of the semantics remain the same or similar, many semantics have been added or revised.

The reference implementation uses the NAL (Network Abstraction Layer) concept to separate the network independent part from the code that is specific to the API and protocols of the underlying layer. The reference implementation uses the sockets API and TCP/IP for its transport mechanism. While this is not overly efficient, the code used to implement Portals 3.3 can be understood by the many people who are familiar with the sockets API. Furthermore, TCP/IP is so widespread that the reference implementation is executable on a large array of machines and networks.

There is a build option that disables a separate progress thread which allows Portals to make progress (sending an *acknowledgment* for example) without the layer above making calls into the portals library. This speeds up the implementation but violates the progress rule.

The source code for the implementation is freely available from the following site:

`ftp://ftp.sandia.gov/outgoing/pub/portals3`

In addition to comments in the code, it contains several README files that describe the implementation. Feedback is highly encouraged to the code author, `jaschut@sandia.gov`, and the Portals 4.0 team at Sandia National Laboratories, `p3@sandia.gov`.

A NAL that runs in Linux kernel space is currently under development.

We maintain a portals web site at http://www.cs.sandia.gov/Portals with links to the latest reference implementation and other information.

D.2 Portals 3.3 on the Cray XT3/XT4/XT5 Red Storm

There are two implementations of Portals available on Cray's XT3/XT4/XT5 Red Storm system. One, generic, is provided by Cray with the machine. The second, accelerated, is under active development at Sandia National Laboratories. There are plans to merge the two versions in the future.

D.2.1 Generic

This is the version provided by Cray with its XT3/XT4/XT5 Red Storm systems. A large portion of the portals code is implemented inside the kernel. When messages arrive at the Seastar NIC, it causes an interrupt and lets the kernel process the portals header; i.e., resolve portal table addressing and match list traversal. The accelerated version under development places more of the portals code inside the Seastar NIC and avoids the interrupt processing on each message arrival.

The generic implementation does not completely match the definitions in the version 3.3 document. The main differences are listed here:

- **PtlHandleIsEqual()** is not implemented.
- **Limitations on IOVECs:** Only the first and last entry can be unaligned (at the head of the buffer and at the tail of the buffer, everything else must be quad-byte aligned).
- There are three new functions that are not part of this document: **PtlIsValidHandle()**, **PtlSetInvalidHandle()**, and **PtlTestAtomic()**.
- The following return codes are not implemented: **PTL_MD_ILLEGAL**, and **PTL_IFACE_INVALID**.
- The type **ptl_size_t** is 32 bits wide, not 64 bits.
- **PtlEQGet()** and **PtlEQWait()** may return a **ptl_event_t** structure that is not fully filled in.

Please refer to Cray documentation for up-to-date information.

D.2.2 Accelerated

An accelerated version that avoids interrupts for each message arrival is being developed and tested at Sandia National Laboratories. At the moment is has more limitations than the generic implementation and leaves out several features discussed in this document. The main ones are:

- Adds a **PtlPost()** call which combines a and **PtlMDUpdate()** call. This eliminates a protection domain boundary crossing in many of the common usage cases.
- The **PtlGet()** operation generates a PTL_EVENT_SEND event.

Since this implementation is still under active development, further changes are to be expected.

Appendix E

Summary of Changes

The most recent version of this document described Portals version 3.3 [Riesen et al. 2006]. Since then we have made changes to the API and semantics of Portals, as well as changes to the document. This appendix summarizes the changes between version 3.3 and the current 4.0 version. Many of the fundamental changes were driven by the desire to reduce the tight coupling required between the application processor and the portals processor, but some additions were made to better support lighter weight communications models such as PGAS.

Foremost, Portals version 4.0 adds a mechanism to cope better with the concept of unexpected messages in MPI. Whereas version 3.3 used PtlMDUpdate() to atomically insert items into the match list so that the MPI implementation could manage unexpected messages, version 4.0 adds an overflow list where the application provides buffer space that the implementation can use to store unexpected messages. The implementation is then responsible for matching new list insertions to items that have arrived and are resident in the overflow list space. This change was necessary to eliminate round trips between the processor and the NIC for each item that was added to the match list (now named the priority list).

A second fundamental change separated all resources for initiators and targets. Memory descriptors are used by the initiator to describe memory regions while list entries are used by targets to describe the memory region *and* matching criteria (in the case of match list entries). This separation of resources was also extended to events, where the number of event types was significantly reduced and initiator and target events were separated into different types with different accessor functions.

In support of the lightweight communication semantics required by PGAS models, lightweight “counting” events and acknowledgements were added. In addition, a non-matching interface was created to decrease the processing required for PGAS messages. Finally, a **PtlAtomic()** function was added to support functionalities commonly provided in PGAS models.

To better offload collective operations, a set of *triggered* operations were added. These operations allow an application to build non-blocking, offloaded collective operations with independent progress. They include variants of both the data movement operations (get and put) as well as the atomic operations.

Anoter set of changes arise from a desire to simplify hardware implementations. The threshold value was removed from the target and was replaced by the ability to specify that a match list entry is “use once” or “persistent”. List insertions occur *only* at the tail of the list, since unexpected message handling has been separated out into a separate list.

Access control entries were found to be a non-scalable resource, so they have been eliminated. At the same time, it was recognized that the `PTL_LE_OP_PUT` and `PTL_LE_OP_GET` semantics required a form of matching. These two options along with the ability to include user ID or job ID based authentication were moved to *permissions fields* on the respective list entry or match list entry.

Index

A

A

B

C

D

E

F

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

T

U

V

W

Z

(*n*) page *n* is in the bibliography.
[*n*] page *n* is in the glossary.
n page of a definition or a main entry.
n other pages where an entry is mentioned.

DISTRIBUTION:

1 Arthur B. Maccabe
University of New Mexico
Department of Computer Science
Albuquerque, NM 87131-1386

1 Trammell Hudson
c/o OS Research
1527 16th NW #5
Washington, DC 20036

1 Eric Barton
9 York Gardens
Clifton
Bristol BS8 4LL
United Kingdom

1	MS 0806	Jim Schutt, 4336
1	MS 0817	Doug Doerfler, 1422
1	MS 0817	Sue Kelly, 1422
1	MS 1110	Ron Brightwell, 1423
1	MS 1110	Neil Pundit, 1423
4	MS 1110	Rolf Riesen, 1423
1	MS 1110	Lee Ward, 1423
1	MS 1110	Ron Oldfield, 1423
1	MS 1110	Kevin Pedretti, 1423
1	MS 1110	Keith Underwood, 1422
1	MS 0899	Technical Library, 9536 (electronic)

Sandia National Laboratories